INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Multicore Scheduling of Real-Time Irregular Parallel Algorithms in Linux

JOÃO PEDRO ALMEIDA BERNARDO FERREIRA
Outubro de 2015

POLITÉCNICO
DO PORTO

Multicore Scheduling of Real-Time Irregular Parallel Algorithms in Linux

João Pedro Almeida Bernardo Ferreira

Dissertação para a obtenção do Grau de Mestre em
Engenharia Informática

Área de especialização em **Sistemas, Arquitecturas e Redes**

Orientador
Doutor Luís Miguel Pinho Nogueira

Júri
Presidente: Doutor x1,
Professor Coordenador no Departamento de Engenharia Informática
do Instituto Superior de Engenharia do Porto
Vogais: Doutor x2,
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 2015

Resumo Alargado

Face à estagnação da tecnologia uniprocessador registada na passada década, aos principais fabricantes de microprocessadores encontraram na tecnologia multi-core a resposta às crescentes necessidades de processamento do mercado. Durante anos, os desenvolvedores de software viram as suas aplicações acompanhar os ganhos de performance conferidos por cada nova geração de processadores sequenciais, mas à medida que a capacidade de processamento escala em função do número de processadores, a computação sequencial tem de ser decomposta em várias partes concorrentes que possam executar em paralelo, para que possam utilizar as unidades de processamento adicionais e completar mais rapidamente.

A programação paralela implica um paradigma completamente distinto da programação sequencial. Ao contrário dos computadores sequenciais tipificados no modelo de Von Neumann, a heterogeneidade de arquiteturas paralelas requer modelos de programação paralela que abstraiam os programadores dos detalhes da arquitectura e simplifiquem o desenvolvimento de aplicações concorrentes. Os modelos de programação paralela mais populares incitam os programadores a identificar instruções concorrentes na sua lógica de programação, e a especificá-las sob a forma de tarefas que possam ser atribuídas a processadores distintos para executarem em simultâneo. Estas tarefas são tipicamente lançadas durante a execução, e atribuídas aos processadores pelo motor de execução subjacente. Como os requisitos de processamento costumam ser variáveis, e não são conhecidos *a priori*, o mapeamento de tarefas para processadores tem de ser determinado dinamicamente, em resposta a alterações imprevisíveis dos requisitos de execução.

À medida que o volume da computação cresce, torna-se cada vez menos viável garantir as suas restrições temporais em plataformas uniprocessador. Enquanto os sistemas de tempo real se começam a adaptar ao paradigma de computação paralela, há uma crescente aposta em integrar execuções de tempo real com aplicações interativas no mesmo hardware, num mundo em que a tecnologia se torna cada vez mais pequena, leve, ubíqua, e portável. Esta integração requer soluções de escalonamento que simultaneamente garantam os requisitos temporais das tarefas de tempo real e mantenham um nível aceitável de *QoS* para as restantes execuções. Para tal, torna-se imperativo que as aplicações de tempo real paralelizem, de forma a minimizar os seus tempos de resposta e maximizar a utilização dos recursos de processamento. Isto introduz uma nova dimensão ao problema do escalonamento, que tem de responder de forma correcta a novos requisitos de execução imprevisíveis e rapidamente conjeturar o mapeamento de tarefas que melhor beneficie os critérios de performance do sistema.

A técnica de escalonamento baseado em servidores permite reservar uma fração da capacidade

de processamento para a execução de tarefas de tempo real, e assegurar que os efeitos de latência na sua execução não afectam as reservas estipuladas para outras execuções. No caso de tarefas escalonadas pelo tempo de execução máximo, ou tarefas com tempos de execução variáveis, torna-se provável que a largura de banda estipulada não seja consumida por completo. Para melhorar a utilização do sistema, os algoritmos de partilha de largura de banda (*capacity-sharing*) doam a capacidade não utilizada para a execução de outras tarefas, mantendo as garantias de isolamento entre servidores.

Com eficiência comprovada em termos de espaço, tempo, e comunicação, o mecanismo de *work-stealing* tem vindo a ganhar popularidade como metodologia para o escalonamento de tarefas com paralelismo dinâmico e irregular. O algoritmo p-CSWS combina escalonamento baseado em servidores com *capacity-sharing* e *work-stealing* para cobrir as necessidades de escalonamento dos sistemas abertos de tempo real. Enquanto o escalonamento em servidores permite partilhar os recursos de processamento sem interferências a nível dos atrasos, uma nova política de *work-stealing* que opera sobre o mecanismo de *capacity-sharing* aplica uma exploração de paralelismo que melhora os tempos de resposta das aplicações e melhora a utilização do sistema.

Esta tese propõe uma implementação do algoritmo p-CSWS para o Linux. Em concordância com a estrutura modular do escalonador do Linux, é definida uma nova classe de escalonamento que visa avaliar a aplicabilidade da heurística p-CSWS em circunstâncias reais. Ultrapassados os obstáculos intrínsecos à programação da kernel do Linux, os extensos testes experimentais provam que o p-CSWS é mais do que um conceito teórico atrativo, e que a exploração heurística de paralelismo proposta pelo algoritmo beneficia os tempos de resposta das aplicações de tempo real, bem como a performance e eficiência da plataforma multiprocessador.

Palavras-chave: Escalonamento de tempo real, sistemas abertos de tempo real, computação paralela, *task-parallelism*, *capacity-sharing*, *work-stealing*, Linux

Abstract

With sequential machines approaching their physical bounds, parallel computers are rapidly becoming pervasive in most areas of modern technology.

To realize the full potential of parallel platforms, applications must split onto concurrent parts that can be assigned to different processors and execute in parallel. Parallel programming models abstract the myriad of parallel computer specifications to simplify the development of concurrent applications, allowing programmers to decompose their code onto concurrent tasks, and leaving it to the runtime system to schedule these tasks for parallel execution. The resulting parallelism is often input-dependent and irregular, requiring that the mapping of tasks to processors be performed at runtime in response to dynamic changes of the workload.

Motivated by the promises of performance scalability and cost effectiveness, real-time researchers are now beginning to exploit the benefits of parallel processing, with ground-breaking scheduling heuristics to improve the efficiency of time-sensitive concurrent applications. Real-time developments are switching to open scenarios, where real-time tasks of variable and unpredictable size share the available processing resources with other applications, making it essential to utilize as much of the available processing capacity as possible.

The p-CSWS algorithm employs *bandwidth isolation*, *capacity-sharing* and *work-stealing* to exploit the intra-task parallelism of hard and soft real-time executions on parallel platforms. This thesis proposes an implementation of the p-CSWS scheduler for the Linux kernel, to evaluate its applicability to real scenarios and bring Linux one step closer to becoming a viable open real-time platform.

To the best of our knowledge we are the first to employ scheduling heuristics to exploit dynamic parallelism of real-time tasks on the Linux kernel.

Through extensive tests, we show that....

Keywords: Open real-time scheduling, parallel computing, task-parallelism, work-stealing, Linux

Acknowledgements

bla bla

Contents

List of Figures

List of Tables

Acronyms

Adeos	Adaptative Domain Environment for Operating Systems
API	Application Programming Interface
BF	Best-Fit
CASH	Capacity-Sharing
CBS	Constant Bandwidth Server
CFS	Completely Fair Scheduler
CGroup	Kernel Control Group
CISTER	Research Centre in Real-Time Computing and Embedded Computing Systems
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecure
DAG	Directed Acyclic Graph
DM	Dealine-Monotonic
DS	p-CSWS Dedicated Server
EDF	Earliest Deadline First
FF	First-Fit
FIFO	First-In First-Out
FP	Fixed-Priority
G-EDF	Global Earliest Deadline First
GPGPU	General Purpose computation on Graphics Processing Units
GPL	GNU General Public License
GPOS	General-Purpose Operating System
GPU	Graphics Processing Unit

HPC	High-Performance Computing
HPP	Heterogeneous Parallel Programming
HRT	Hard Real-Time
IPC	Inter-Processor Communication
ISR	Interrupt Service Routine
JLFP	Job-Level Fixed-Priority
LIFO	Last-In First-Out
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MMU	Memory Management Unit
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
ORT	Open Real-Time
OS	Operating System
p-CSWS	Parallel Capacity Sharing by Work-Stealing
PAS	Priority-Aware Stealing
PGAS	Partitioned Global Address Space
PI	Priority Inheritance
QoS	Quality of Service
RCS	p-CSWS Residual Capacity Server
RM	Rate-Monotonic
RR	Round-Robin
RTAI	Real-Time Application Interface
RTLinux	Real-Time Linux
RTOS	Real-Time Operating System
RTS	Real-Time System
RTWS	Real-Time Work-Stealing
SCI	System Call Interface
SIMD	Single Instruction, Multiple Data

SISD	Single Instruction, Single Data
SMP	Symmetric Multiprocessing
SPMD	Single Program, Multiple Data
SRT	Soft Real-Time
UMA	Uniform Memory Access
VFS	Virtual File System
WCET	Worst-Case Execution Time
WS	Work-Stealing

Chapter 1

Introduction

"There is nothing more dangerous than not taking a risk."

— Pep Guardiola

1.1 Motivation

The undeniable benefits of multiprocessor platforms are currently instigating a paradigm change in several areas of computer science, including real-time research. Leading the recent outbreak in ubiquitous computing and mobile devices, multi-core machines are quickly becoming prominent as the industry tries to keep up with the performance and power consumption demands of modern applications.

To exploit the advantages of multiprocessor technology, a multitude of parallel programming frameworks now allow developers to slice their programming logic into execution threads that may run in parallel across the available processing units. Although flexible and easily expressed, parallelism of this nature is often irregular, and relies on the underlying process scheduler dynamically look for parallel execution.

With the ever-increasing interest in time-sensitive user-end applications, such as continuous media streaming, research is now focusing on bringing real-time support to the mainstream, to provide General Purpose Operating Systems with feasible scheduling solutions for real-time tasks competing for processor time with traditional desktop applications. Open Real-Time (ORT) systems of this kind should achieve a correct allocation of the available resources so that (i) the requirements of real-time computations are met, and (ii) the overall performance for other applications is kept at a satisfactory level.

Deterministic real-time environments focus on Worst-Case Execution Time (WCET) analysis to guarantee the execution requirements of critical hard real-time tasks. Precise estimation of the WCET is usually difficult to obtain, and a pessimistic outlook on the real execution requirements of a given task, ultimately leading to inefficient use of the available resources. Time-sensitive computations are no longer restricted to control routines or safety-critical computations,

and WCET scheduling is inadequate for ORT scenarios, where execution times are expected to be particularly variable. However, in order to achieve better utilization of the available processing bandwidth, it is possible to schedule non-critical soft real-time tasks by their mean execution values, as long as an acceptable degree of Quality of Service (QoS) is maintained. Bandwidth reservation mechanisms allow soft real-time tasks to occasionally overlook their timing constraints, by providing temporal isolation, and preventing overrun effects to compromise the schedulability of other tasks.

Contrarily to early dedicated systems, ORT environments should be prepared to deal with unpredictable amounts of work, and as the volume of the computation grows, scalability calls for multiprocessor architectures. With SCHED_DEADLINE recently making it into the mainline kernel, Linux is now able to manage irregular real-time applications using the Constant Bandwidth Server (CBS) abstraction, but even though multi-core scheduling is contemplated, it is still unable to deal with intra-task parallelism. While other remarkable approaches such as tackled Linux support for intra-task parallelism, they are designed exclusively for hard real-time computations scheduled by the WCET, and do not address the problem of irregular workloads.

The Parallel Capacity Sharing by Work-Stealing (p-CSWS) scheduler combines bandwidth reservation with Capacity-Sharing (CASH) and Work-Stealing (WS), to efficiently schedule parallel real-time tasks of unpredictable and variable workloads. This thesis proposes a new scheduling module for the Linux kernel based on the p-CSWS algorithm. To the best of our knowledge, we are the first to take on dynamic multi-core scheduling of irregular parallel real-time tasks in the Linux kernel, and provide a scheduling solution that meets the basic requirements of a multiprocessor ORT system.

1.2 Contributions

This project breaks new ground as it is the first to explore the practicality of scheduling both hard and soft parallel real-time tasks in the Linux kernel. Based on the theoretical proposal of the p-CSWS scheduler, we introduce a new scheduling class capable of:

- Dynamically performing multi-core scheduling of real-time computations
- Handling unexpected load changes in irregular and aperiodic tasks
- Exploiting intra-task parallelism in real-time executions

With this document, our main goal is to prove the effectiveness of our implementation through the analysis of extensive experimental results.

The work described in this document has been published in [Ferreira and Nogueira, 2013]. A journal version is under submission.

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 Sys-

tems). 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, focusing on the analysis, 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 ranking was confirmed in the last evaluation on 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

This document is structured as follows:

- Chapter 2 presents a study of parallel computer architectures, programming models, and dynamic scheduling of parallel tasks.
- Chapter 3 provides an in-depth study of real-time systems, defining the concept of real-time applications and surveying real-time scheduling techniques including the p-CSWS scheduler serving as the theoretical foundation of this thesis.
- Chapter 3 studies the Linux kernel, with special emphasis to the internals of the process scheduler, and lists related work on Linux real-time support.
- Chapter 5 provides a detailed description of the SCHED_PCSWS scheduler and its implementation.
- Chapter 6 assesses the efficiency and correctness of the SCHED_PCSWS scheduler.
- Chapter 7 concludes and suggests future extensions to our work.

Chapter 2

Parallel Computing

> *"[...] a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs."*
>
> — Edward A. Lee

The past decade marked a turning point with regard to the adoption of parallel processing platforms in the general market. For many years, the industry tried to attend the performance needs of user-end sequential programs by improving clock speed and efficiency with each new generation of single-core processors [Diaz et al., 2012]. As the inherent limitations of the architecture became evident and single-core evolution became stagnant, it was realized that performance scalability could be achieved more efficiently through the combination of several processing units, or cores, in the same chip. The multi-core paradigm proved beneficial for both ends of the market, as manufacturers were able to design increasingly powerful solutions at a reduced cost, and significantly decrease clock speed and energy consumption to improve cost-efficiency for the customer.

Despite the recent outbreak in the domestic and small-business markets, parallel computing has long had an important role in diverse areas of computer science, from large industrial applications to the vast domain of small embedded devices. Since there is no upper bound on the number of microprocessors that can be interconnected, parallel architectures offer the theoretical promise of limitless processing capacity [Dongarra et al., 2003]. This is the idea behind most developments in the High-Performance Computing (HPC) niche, where multiple loosely-coupled processors are combined to handle unusually large workloads.

However, in contrast to uniprocessor machines unified by the Von Neumann model, there exist many ways to couple individual Central Processing Units (CPUs) in a single computing system. As a result, the structural design and *modus operandi* of parallel computers can be rather heterogeneous. On the other hand, while parallel computers are able to provide a virtually infinite amount of computing power, they are unquestionably harder to program and use [Dongarra et al., 2003, mei Hwu et al., 2008].

We begin this chapter with a broad survey on distinct types of parallel computers and archi-

tectures. We will then move on to parallel software, referencing the particularities of parallel programs as opposed to sequential algorithms, as well as addressing different models of parallel programming. We conclude with a quick view on dynamic scheduling of task-parallel applications.

2.1 Parallel Computer Architectures

In the general case, it is simply not practical to study or operate computers at the machine level. To make productive and convenient use of computers, technologists rely on archetypal *models of computation*, which reduce their structural and functional complexity to a small set of abstract components and their relations [Kumar, 2002, Navarro et al., 2014]. A model of computation is a conceptual machine that outlines the essential properties of a certain class of computers sharing a set of common characteristics. Through a comprehensive characterization, it unifies several configurations into a single abstract platform, and provides a means of evaluating the theoretical performance of algorithms on the same platform. [Hambrusch, 1996, Kessler and Keller, 2007, Maggs et al., 1995, Skillicorn and Talia, 1998]

Models of computation can be divided into two interdependent parts. The *programming model* describes the basic operations supported by a computer, defining a set of programming abstractions that can be used to derive machine-independent programming languages. This allows easy formulation and understanding of portable algorithms, as it enables developers to program the abstract platform rather than a specific computer. The *architectural model*[1] appears at a lower level of abstraction, to establish the bridge between the programming model and the hardware platform. It specifies the execution engine, characterizing each operation in terms of the abstract components involved, their interactions, and the respective cost. Such description of the operational costs makes it possible to infer the combined cost of entire algorithms and analyze their performance on the computing platform. [Hambrusch, 1996, Kessler and Keller, 2007, Maggs et al., 1995, Skillicorn and Talia, 1998]

Parallel computing alludes to the simultaneous and cooperative use of multiple processing units to solve a computational problem [Foster, 1995]. The maturity reached by sequential computers over the years is partly owed to the simplicity and robustness of the conceptual Von Neumann machine [Goldstein and von Neumann, 1961], widely accepted as the standard model of computation for serial computers [Foster, 1995, Kessler and Keller, 2007, Navarro et al., 2014]. Such maturity is yet to be attained in the parallel computing domain where, due to the heterogeneity of computer designs, several models of parallel computation have been proposed in literature.

Parallel computer classifications are based upon varied differentiation criteria. In the remainder of this Section, we study the leading architectural models of parallel computation. Parallel programming models will be presented in Section 2.3.

[1] Also called *machine model*, *cost model*, *performance model*, or *hardware model*.

2.1.1 Flynn's Taxonomy

According to Flynn [1972], instructions and data make up the two types of information that flow into a processor. A succession of related instructions is referred to as an *instruction stream*, while a *data stream* is a sequence of data elements affected by an instruction stream.

The popular classification of Flynn is based upon the plurality of instruction and data streams in effect simultaneously. The combination of scenarios yields 4 distinct architectures.

- *Single Instruction, Single Data (SISD)* - A uniprocessor architecture executing a single instruction upon a single data element, at each clock cycle. The SISD abstraction directly translates into the Von Neumann architecture. [El-Rewini and Abd-El-Barr, 2005, Kshemkalyani and Singhal, 2008, Navarro et al., 2014].

- *Single Instruction, Multiple Data (SIMD)* - A type of parallel computer in which several processing units synchronously execute the same instruction on different sets of data (*data parallelism*). It is typical of systems performing recurrent computation, such as graphics processing, or as a redundancy mechanism in fault-tolerant platforms.

- *Multiple Instruction, Single Data (MISD)* - Another parallel architecture, whose processing units execute different instruction streams on a single data element. MISD is not considered very practical, though it may be useful for complex problem-solving algorithms.

- *Multiple Instruction, Multiple Data (MIMD)* - Models the majority of parallel computers available today. In a MIMD computer, each processing unit is asynchronously controlled by distinct instruction streams operating on different elements of data.

The highly abstract models of Flynn sort computers by behavioral paradigm, but do not provide a thorough description of their architectural design and execution engine.

Parallel computers are broadly described as an arrangement of 3 components: (i) a processing module comprised of several CPUs, (ii) the main memory, and (iii) the interconnection network [Kumar, 2002, Skillicorn and Talia, 1998]. The physical organization of these components, particularly the spacial relationship between processors and memory, is the central differentiating factor among parallel computer architectures, and one that has a direct impact on their operation and performance.

Owing to their prevalence in the consumer market, in Section 2.1.2 we focus on MIMD computers, their memory architectures, and communication models.

2.1.2 Parallel Memory Architectures

If several processors are to work cooperatively in the resolution of a problem, they must communicate either by direct manipulation of a shared memory space or by message exchanging through the interconnection network. MIMD machines can be further classified by the adopted communication model supported by the underlying hardware design. [Dongarra et al., 2003, El-Rewini and Abd-El-Barr, 2005]

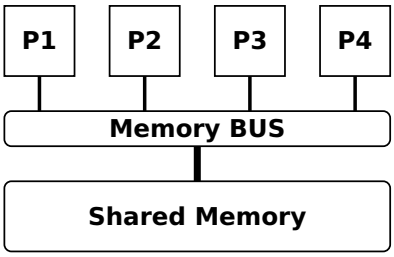

Figure 2.1: Shared memory computer.

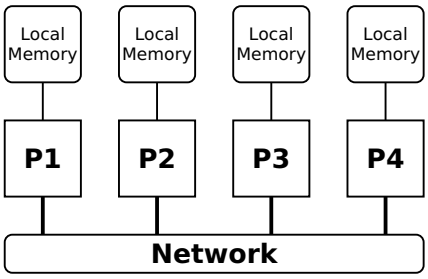

Figure 2.2: Distributed memory computer.

2.1.2.1 Shared Memory Architectures

A *shared memory* architecture (Figure 2.1) is a collection of *tightly-coupled* processing units, linked to a single memory resource by way of a common interconnection network[2]. Inter-Process Communication (IPC) is achieved through concurrent and coordinated access to a common set of data in the global memory space. Each processor manipulates the shared data independently, and each change is effective and visible to all processors. [Dongarra et al., 2003, El-Rewini and Abd-El-Barr, 2005]. In literature, shared memory computing resources are commonly referred to as *multiprocessors* [Kshemkalyani and Singhal, 2008, Quinn, 2004]. As a collection of several processing units sharing a single memory space, a commodity *multi-core* configuration falls into this category.

At the hardware level, cache coherency is the most problematic aspect of this model. A cache coherent system is one which guarantees equality between cache and memory values, across processors, at all times. Under this perspective, it must be assured that programs execute as if they were accessing the memory directly, usually through additional hardware mechanisms[3] specifically designed for this purpose. [Dongarra et al., 2003]

With respect to cache coherency and memory access times, shared memory models can be further separated into two main categories, namely *Uniform Memory Access (UMA)* and *Non-Uniform Memory Access (NUMA)*.

In UMA machines, all processors access the memory uniformly at the same speed. In modern implementations, processors, cache, and main memory, are directly interconnected to guarantee cache coherency and equal access times to memory by all processors. However, UMA systems lack in scalability, due to the exponential growth of the interconnection network and the bottleneck inherent to memory contention on a single bus [Dongarra et al., 2003, Navarro et al., 2014, Nitzberg and Lo, 1991].

Conversely, NUMA architectures exhibit variable access times, depending on the location of processors in relation to the memory. A NUMA configuration consists of several tightly-coupled multiprocessors, each with a dedicated memory space, and connected through a scalable network[4] [El-Rewini and Abd-El-Barr, 2005, Wilson, 1987]. Communication is achieved by memory address mapping, so that each individual processor can directly access and manipulate

[2]Typically a bus.
[3]Known as *cache-coherency protocols* [Dongarra et al., 2003].
[4]Such as a tree, or a hierarchical bus.

the memory dedicated to another processor. Even though a NUMA system can scale to much higher numbers of processors and capacity than a UMA model, memory access between links is generally slower [Kshemkalyani and Singhal, 2008].

The combination of multi-core processors with many-core Graphics Processing Units (GPUs) eligible for general-purpose computation (General Purpose computation on Graphics Processing Units (GPGPU)), defines yet another type of multiprocessor configuration where highly parallel applications can achieve great speedup. While this type of architecture falls into the NUMA category, *AMD* recently began to tackle a solution where the main memory is shared equally among the CPU and GPU, in an attempt to minimize communication latency, and bring the CPU and GPU together as a single unit in a programming platform [Kyriazis, 2013].

2.1.2.2 Distributed Memory Architectures

Distributed memory architectures (Figure 2.2), usually consist of multiple independent computers, called *nodes*, connected though a network, and using this medium to exchange messages when communication across nodes is desired [Dongarra et al., 2003, El-Rewini and Abd-El-Barr, 2005]. As collections of multiple sequential or parallel computers, distributed memory configurations are also known as *multicomputers* [Kessler and Keller, 2007, Kshemkalyani and Singhal, 2008, Quinn, 2004].

In distributed memory architectures, there is no notion of a global address space. The existing memory is privately owned by each computer, isolated from the network, and kept inaccessible to programs executing in other nodes, so that any memory manipulations are performed locally. Seeing that each machine has exclusive access to its own memory space, the data consistency and cache coherency setbacks of shared memory architectures do not hold [El-Rewini and Abd-El-Barr, 2005].

Arguably the greatest advantage of distributed memory computers, lies in that they can theoretically scale to infinite numbers. Since memory space can increase proportionally to the number of processors, scalability is only limited by the topology of the network. For this reason, large computer clusters and grids are typical examples of distributed memory implementations [Dongarra et al., 2003].

However, distributed memory programming through message-passing introduces a number of facets that must be addressed by the developer. Because the data is spread across independent machines, *locality awareness* is crucial for the communication procedure, as each program must know exactly where other programs, along with the respective data, are located within the system [Jordan, 1991, Kumar, 2002].

Programmers must also acknowledge the communication overhead associated with message-passing, *i.e.* the costs of constructing, interpreting, and transmitting each message through the interconnection network [Jordan, 1991]. These costs, along with the communication bottleneck imposed by the interconnection network, suggest that parallel programs requiring a minimal amount communication are the better suited for the distributed memory paradigm. As such, it is advisable that the workload be partitioned and distributed in such a manner that data locality is maximized

and communication is reduced to a minimum.

2.1.2.3 Hybrid Distributed-Shared Memory Architectures

Advancements in the area of HPC promote alternative archetypes, which combines shared and distributed memory into hybrid architectures. Ingeniously conceived *distributed-shared memory* configurations sustain the largest and fastest supercomputers of today [Diaz et al., 2012]. Hybrid architectures are prominent in many *cutting-edge* fields of modern computing technology, such as cloud computing or social networking.

The obvious, but challenging goal of distributed-shared memory is to join the advantages of both architectures, especially the scalable potential of distributed memory, and the superior efficiency and simplicity of the shared memory communication model.

Hybrid architectures introduce the concept of *virtual shared memory*, with a simulated global address space that abstracts the distributed-memory configuration and guarantees transparent access to the data, regardless of its location. At a large scale, the costs of storage and communication, as well as many performance aspects, are influenced by the adopted data distribution scheme, for which two main techniques are used: *data migration* and *data replication*. Data migration offers an efficient storage solution, at the expense of increased volumes of communication. Data replication looks to reduce communication and improve performance, through a model of redundant data distribution that often degrades consistency. [Dongarra et al., 2003, Nitzberg and Lo, 1991, Protic et al., 1997]

Data distribution is merely one of numerous dilemmas in the discipline that tackles the design and implementation of distributed computing platforms, where the tradeoff between consistency, availability, and partition tolerance, as described by Brewer's *CAP Theorem* [Brewer, 2000], plays a dominant role. As interesting as it is to examine the plethora of challenges and *state-of-the-art* developments in this area, we believe that they extend far beyond the scope of our work.

2.2 Towards Parallelism

Augmenting the number of processors raises the level of processing power and the ability to handle increasingly larger workloads, but does not directly translate into performance enhancement. Think of two independent sequential programs, requesting processor time in a uniprocessor architecture. If the uniprocessor was to be replaced by two microprocessors of the same capacity, doubling the amount of processing units would allow for both programs to run simultaneously and improve the overall response-time of the system, but the performance of each individual program would not increase accordingly.

To better utilize the available processing capacity and scale application efficiency, computations must parallelize. Programs must be parceled out into smaller execution chunks that can be distributed, and coordinately migrated between processors, to execute in parallel. Ultimately, as long as there exists work awaiting completion, all processing units should be kept busy at every clock cycle to exploit the maximum amount of conjoint processing power.

2.2.1 Concurrency as the Key Towards Parallelism

In simple terms, *speedup* measures the performance gain achieved by solving a computational problem in parallel, rather than sequentially. As expressed by Equation 2.1, the speedup of a program is given by the ratio of sequential execution time $T(1)$ on a single processor, to parallel execution time $T(n)$ on a parallel platform of n identical processors.

$$Speedup(n) = \frac{T(1)}{T(n)}. \tag{2.1}$$

Amdahl's Law, deducted from the observations of Gene Amdahl in Amdahl [1967], uses this definition to show how the the growth in performance of a program, as it switches from a single processor to a parallel configuration, is constrained by its sequential portion [5]. From another perspective, the attainable theoretical speedup hinges upon the amount of work that can be executed in parallel, *i.e.* the concurrent portion of the program.

Because the concepts *concurrency* and *parallelism* can be deceptive, we subscribe to the definition of concurrency as a property that enables distinct parts of an algorithm to execute simultaneously, and parallelism as the actual event where two or more sections of a program execute simultaneously. In this sense, parallelism is said to be a subclass of concurrency [Navarro et al., 2014] . Throughout this document, we may use the terms *concurrent* and *parallel* interchangeably, in reference to concurrent programs eligible for parallel execution.

Amdahl assumes that the concurrent portion of a program is perfectly parallel[6] and remains constant throughout execution, but such assumptions fall far from practical truth [Hill and Marty, 2008]. For instance, if concurrent sections of a program contend for the same memory resource, synchronization mechanisms must serialize the access to ensure data consistency. Resource contention is only one of several sources of serialization overhead that threaten the amount of achievable parallelism, and the performance of parallel executions is influenced by a multitude of aspects that are not taken into account by Amdahl's observations. For these reasons, although a fine theoretical remark, Amdahl's law is not directly applicable to real settings [Hill and Marty, 2008, Kumar, 2002, Roth et al., 2012].

Nevertheless, seeing concurrency as the key prerequisite for optimizing the performance of individual applications and the efficiency of parallel systems, there is an emphasis on improving the exploitable concurrency of algorithms, in an attempt to offer the best opportunities for parallelism.

2.2.2 Finding and Expressing Concurrency

Unfortunately, the vast majority of computer software available today follows a sequential instruction flow unsuitable for parallel execution. To harness the maximum amount of processing capacity, developers are now focusing on the design and development of parallel applications either by adjusting existing algorithms, or creating entire parallel programs. In fact, these are the

[5] In parallel computing literature, this limitation is referred to as *serial bottleneck*

[6] Concurrent parts of the same program never serialize throughout execution.

two primary methods of producing concurrent algorithms: *automatic parallelization* and *parallel programming*. [Diaz et al., 2012, Dongarra et al., 2003]

Automatic parallelization refers to the use of specialized tools that take sequential computing logic and automatically recognize opportunities for parallelism, resulting in what is called *implicit parallelism* [Dongarra et al., 2003, Kumar, 2002, Navarro et al., 2014]. The serial program is often decomposed by a compiler or interpreter, able to identify well-known patterns and split the logic and data into concurrent parts. Though a simple solution to adapt existing sequential code, automated code analysis is complex, rudimentary, and unlikely to produce powerful and efficient parallel algorithms in the general case [Diaz et al., 2012, Dongarra et al., 2003].

Highly efficient parallel applications can be built through explicit parallel programming practices, relying on the expertise of the software developer to formulate concurrency. *Explicit parallelism* [Dongarra et al., 2003, Kumar, 2002] of this kind requires the programmer to decompose the computation into concurrent units of control and manage their interactions. Expressing parallelism through explicit programming can be an intricate and laborious process, with many engaging facets that remain widely debated across software engineering. [Dongarra et al., 2003, Kumar, 2002]

2.2.2.1 Problem Decomposition

The key to parallel programming lies in identifying and exposing the exploitable concurrency within the computational problem. While the ordinary challenges of sequential programming persist, developers are faced with the supplementary effort of partitioning the problem into concurrent pieces and managing the dependencies among them. This process, called *problem decomposition*, or *problem partitioning*, is classified by the renowed methodological approach of Foster [1995] as the primary stage of parallel algorithm design.

In the decomposition strategy known as *task decomposition*, or *functional decomposition*, the initial focus is upon the computation, as concurrent instruction sequences are identified and apportioned into individual tasks. A *task* is a unit of control, with its own data space, that can be allocated to the processor by a scheduler. [Breshears, 2009, Dongarra et al., 2003, Foster, 1995, Rauber and Rünger, 2010]

Once the computation has been broken into tasks, partitioning of the data operated by each task follows. Unfortunately, a decomposition into purely independent tasks is very unlikely. If a computation requires data bound to another task, then both tasks need interact coordinately, through the underlying communication subsystem, to exchange such data. Task dependencies, as such, must also be determined, and either resolved or removed, at the decomposition stage [Breshears, 2009, Dongarra et al., 2003, Rauber and Rünger, 2010].

In other cases, an initial decomposition of the data may be more suitable. Problems of this type typically compute data iteratively, as in adding the values in each row of a two-dimensional array, word counting in a stream of input text, *et cetera*. With *data decomposition*, or *domain decomposition*, the computational problem is divided into a series of smaller ones. The data are partitioned into smaller fragments, and the operations affecting each of these pieces are decom-

posed from the serial computation and organized into individual tasks, which are computed in parallel and synchronize at the end. As with task parallelism, data requirements among the originated tasks may not be thoroughly disjoint. In the event that operations demand data from other partitions, the communication subsystem must be put into service to make such data available to the requesting task. [Breshears, 2009, Dongarra et al., 2003, Foster, 1995, Kumar, 2002, Rauber and Rünger, 2010]

When applied to the class of problems exemplified above, a data decomposition approach usually yields identical tasks, set to operate equal-sized data segments of the same structure[7]. As these tasks are distributed across processing units, all processors execute the same program, in parallel, but upon different streams of data. As explained in Section 2.2.2.2 this has been formalized into the prominent Single Program, Multiple Data (SPMD) execution pattern.

2.2.2.2 Parallel Programming Patterns

With task decomposition, programmers merely point out concurrent sections of code, leaving it to the underlying runtime environment to decide whether to launch concurrent tasks in parallel or not. To provide the best flexibility, they are allured to expose as much concurrency as possible, in a type of dynamic parallelism that is often input-dependent, variable, and unpredictable. Since many execution details are not known *a priori*, the problem of mapping tasks to processors must be tackled dynamically, at runtime, to maintain an even distribution of work across the parallel platform [Belikov et al., 2013, Kumar, 2002]. Parallel computing literature refers to this execution paradigm, in which compiled programs dynamically generate tasks at runtime, as *task parallelism* or *intra-task parallelism* [Belikov et al., 2013, Dongarra et al., 2003, Kumar, 2002, Navarro et al., 2014].

Thread-based parallelism, or *multithreading*, is the most common type of task parallelism. In this context, a *thread* is a lightweight stream of control that performs a fraction of the work contained in a program [Kumar, 2002], and the smallest unit recognizable by a scheduler. Threads are generated within the control flow of a *master task*, which can be an independent process or another thread. In addition to the local data, they inherit and share resources allocated to the master task, and use the shared memory region to communicate via asynchronous load and store operations [Diaz et al., 2012]. [Belikov et al., 2013, Breshears, 2009, Diaz et al., 2012, Dongarra et al., 2003, Kumar, 2002, Navarro et al., 2014]

Execution in the *fork/join* style begins with a *sequential region*, as the master task corresponding to the main parallel program is attributed to the CPU. The master task can enter a *parallel region* at any time, as it spawns concurrent child threads using a *fork* statement. Throughout the parallel region, threads and the master task compete against each other[8] for CPU time. A *join* statement instructs the master task to wait for the completion of all child threads. As the last pending thread completes, the master task resumes executing the sequential portion of the program. [Kessler and Keller, 2007, Mattson et al., 2004, Quinn, 2004, Rauber and Rünger, 2010]

[7] For instance, if a two-dimensional array is partitioned into rows, then an equal number of identical tasks can be assigned to sum the elements of each row in parallel.

[8] And possibly other tasks in the system.

Alternatively, if the data are decomposed into independent parts distributed evenly among processors we allude to *data parallelism*, typified by the Single Program, Multiple Data (SPMD) execution pattern [Dongarra et al., 2003, Rauber and Rünger, 2010]. With SPMD, each processor performs the same task upon its own subset of the data. Data and task mapping are performed at configuration time and remain constant until completion, making for a runtime behavior that is substantially more deterministic than that of task parallelism. Here, the keys to a good distribution are load balance and data locality [Kumar, 2002], to avoid the two major sources of inefficiency in this model, namely processor idling and communication overhead. Since parallelism remains unchanged throughout execution, although the quest for these goals can be complex, task interactions can be determined beforehand and taken into account in the design of an efficient distribution scheme. [Belikov et al., 2013, Dongarra et al., 2003, Kessler and Keller, 2007, Kumar, 2002, Mattson et al., 2004, Navarro et al., 2014, Rauber and Rünger, 2010]

Throughout this document we will direct our attention to the task-parallel fork/join model and its derivatives, overwhelmingly popular in shared memory architectures typical of commodity computers [Diaz et al., 2012, Quinn, 2004]. Alternative styles found across literature, such as *master/worker*, *client/server*, *loop parallelism*, *pipelining*, *task pools*, *et cetera* [Rauber and Rünger, 2010], are mostly applicable to particular cases that are of little interest to us. For a comprehensive study of parallel execution and programming patterns, we refer the reader to the specialized book of Mattson et al. [2004] and the annexed bibliography [Diaz et al., 2012, Quinn, 2004, Rauber and Rünger, 2010].

2.2.3 Granularity and its Effect on Performance

So far, we have discussed the importance of identifying and revealing the intrinsic concurrency within a computational problem, as it is set to be solved on a parallel platform. Achieving maximum throughput in a parallel system is primarily a matter of compromise between load-imbalance and parallel overhead [Kshemkalyani and Singhal, 2008], directly influenced by the relationship between the underlying architecture, and the number and size of parallel tasks involved in the computation, *i.e.*, the *granularity* of the parallel program. [Kshemkalyani and Singhal, 2008, Kumar, 2002, Navarro et al., 2014]

Parallel algorithms alternate between periods of computation and periods of synchronization. The volume and frequency of interactions is bound to grow as the volume of concurrent tasks increases and their size diminishes. Thus, by measuring the decomposition in terms of the number of tasks and their length, granularity also quantifies the ratio of computation to communication within the program. [Breshears, 2009, Kshemkalyani and Singhal, 2008, Rauber and Rünger, 2010]

Fine-grained programs, parceled into large numbers of small tasks, offer the best promise of speedup, load balance, and scalability, but are also susceptible to runtime overhead. With too fine a level of granularity, the predictably large communication overhead, together with the costs of bringing parallel tasks to execution and maintaining a balanced distribution of work, may compromise or even supersede the gains of parallel execution and the time expended in

the execution of the program [Breshears, 2009, Kshemkalyani and Singhal, 2008, Rauber and Rünger, 2010]. In these cases, the level of granularity can be adjusted with task *agglomeration*, by grouping concurrent tasks into larger serial segments to reduce operational costs [Foster, 1995, Navarro et al., 2014, Quinn, 2004].

The scenario delineated above is consistent with the description of task-parallel environments presented in Section 2.2.2.2. As communication-intensive algorithms, fine-grained programs are better suited for tightly-coupled architectures, optimized for IPC through a shared memory medium. [El-Rewini and Abd-El-Barr, 2005, Kshemkalyani and Singhal, 2008]

In turn, *coarse-grained* algorithms are partitioned into fewer and larger tasks, which are less prone to interact. Computation-intensive programs as such are expected to outperform fine-grained applications in loosely-coupled machines, where communication across nodes is significantly more expensive. On the other hand, load-balancing flexibility is hampered by the considerable size of sequential computations. [Breshears, 2009, El-Rewini and Abd-El-Barr, 2005, Kshemkalyani and Singhal, 2008, Rauber and Rünger, 2010]

2.3 Parallel Programming Models

As one switches from a unified model of sequential computers to a variety of parallel architectures, a radical shift in perspective is imposed, and the design and development of efficient machine-independent applications becomes markedly more challenging. To alleviate the programmer from architecture specifics, parallel programming models look to offer an efficient cross-platform abstraction for parallel computers, through an assortment of components to design and develop parallel algorithms. Though not tied to a specific platform, and expected to provide compatibility with any computer, parallel programming models are commonly in tune with a particular parallel architecture and the respective communication model. [Diaz et al., 2012, Dongarra et al., 2003]

2.3.1 Shared memory models

The task-parallel pattern is facilitated in shared memory architectures, where part of the communication is implicitly specified by concurrent and direct manipulation of a global memory space [Diaz et al., 2012]. In light of their proximity to sequential computers, shared memory machines are considered the simplest to program, as they rely on normal variable assignments, rather than explicit communication operations, to transmit data between tasks. However, they are not exempt of idiosyncratic issues demanding special knowledge and attention to detail.

Asynchronous manipulations upon a shared memory resource raise data consistency issues that may jeopardize algorithm correctness. On this account, shared memory models provide specialized control and synchronization features[9] to resolve non-deterministic access conflicts and coordinate race conditions between concurrent tasks. While explicit control of concurrency can be notoriously cumbersome and problematic, it is also a source of unpredictable runtime overhead.

[9] Such as mutexes, locks, critical section directives, condition variables, semaphores, barriers, *et cetera*.

Another issue lies with the potential for *deadlocks*, originated when two or more concurrent tasks are each waiting for locks held by the other to be released. [Belikov et al., 2013, Diaz et al., 2012, Dongarra et al., 2003]

POSIX Standard Thread Library (Pthreads) is a popular C programming library, providing a set of low-level specifications to create, destroy, and coordinate parallel threads under the fork/join style. As a very explicit model, it is primarily used by experienced developers, well versed in parallel programming and execution problematics, to build efficient parallel applications at the cost of extensive expertise and development effort. [Diaz et al., 2012, Dongarra et al., 2003]

Open Multi-Processing (OpenMP) [ARB] is the *de-facto* standard for shared memory parallel programming under C, C++ and FORTRAN. Though based on the Pthreads model [Navarro et al., 2014], it offers a portable and cross-platform combination of library routines and compiler directives, to write well structured high-performance parallel applications at a higher level of abstraction [Diaz et al., 2012, Dongarra et al., 2003]. Involving both the programmer and compiler in the decomposition phase, OpenMP is regarded as a partially-implicit model [Navarro et al., 2014]. Synchronization and concurrency control are greatly simplified with application-oriented clauses that complement data and algorithm definition [Dongarra et al., 2003].

2.3.2 Message-Passing Models

Message-passing is a widely accepted view of parallelism for distributed memory architectures, in which tasks own a private memory space and interact via message exchanging through the underlying interconnection network. Here, the data consistency complications of shared memory do not hold, and task synchronization is implicit to each message-passing operation. On the other hand, while programmers need not enforce mutual exclusion, they must explicitly transmit the data with matching *send* and *receive* commands on both ends of the interaction. Although message-passing applications can run efficiently on shared-memory configurations, they naturally match multicomputers, data parallel applications, and the SPMD execution pattern. [Dongarra et al., 2003, El-Rewini and Abd-El-Barr, 2005, Foster, 1995, Kshemkalyani and Singhal, 2008, Kumar, 2002, Quinn, 2004, Rauber and Rünger, 2010]

The standard *Message Passing Interface (MPI)*, defines a rich set of portable library routines to write large applications under the cooperative[10] message-passing model. MPI is paired with a wide variety of programming languages through *language bindings* providing language-oriented syntax [Dongarra et al., 2003, Kumar, 2002]. Despite the variety of routines specified by MPI, as little as 6 are needed to address a multitude of problems [Dongarra et al., 2003, Kumar, 2002].

With the advent of distributed memory clusters of multiprocessor nodes, later versions of MPI were adapted to handle both distributed and shared memory architectures seamlessly, although the inherent paradigm and semantics still lean towards distributed memory. [Diaz et al., 2012]

[10] Also called point-to-point. A *send* operation in a process must be met with a *receive* operation in another process.

2.3.3 Other models

As discussed in Section 2.1.2.3, high-performance computers do not strictly follow the distributed memory approach. To exploit the advantages of both the shared and distributed memory paradigms, modern supercomputers are made of numerous multi-core machines connected through a scalable network. [Diaz et al., 2012]

Although parallel programming models are architecture-independent, programmers usually prefer to blend multithreading techniques with message-passing directives, in a process that often involves a combination of programming models [Diaz et al., 2012].

Another option is to use the *virtual shared memory* abstraction to simulate a global address space over a distributed memory configuration. A solution in vogue is the Partitioned Global Address Space (PGAS), which introduces a locality-aware memory management layer with a clear distinction between local and remote data [Diaz et al., 2012]. Whereas local memory can be accessed with standard sequential mechanisms, remote one-sided access is accomplished via specific PGAS constructs. With a correct distribution of data, applications can be optimized to operate local memory with reduced communication overhead. [Belikov et al., 2013, Diaz et al., 2012, Dongarra et al., 2003]

In the design and development of highly parallel algorithms for GPGPU computation, specialized Heterogeneous Parallel Programming (HPP) models are used. In this field, NVIDIA's Compute Unified Device Architecure (CUDA) and the more compatible Open Computing Language (OpenCL) are the most prominent choices. While OpenCL natively supports CPU and GPU programming, CUDA is frequently combined with multithreading models to write heterogeneous applications with a high degree of parallelism. [Belikov et al., 2013, Diaz et al., 2012]

2.4 Dynamic Scheduling and Load-Balancing

Once computations and data have been partitioned into concurrent tasks, a correct assignment of these tasks to the available processing resources is necessary, to make efficient use of the parallel hardware and maximize the performance of the concurrent program by executing it in parallel. [Belikov et al., 2013, Foster, 1995, Kessler and Keller, 2007, Quinn, 2004]

Scheduling is the process by which tasks are mapped to physical processing units. The performance of parallel configurations is a function of two conflicting sources of inefficiency: (i) inter-processor communication, and (ii) processor idling caused by load-imbalance. The trade-off between both is dictated by the allotment of tasks to processors and the order in which tasks are selected for execution, *i.e.* the *schedule*. [Blumofe and Leiserson, 1993, Foster, 1995, Kessler and Keller, 2007, Narlikar and Blelloch, 1998, Quinn, 2004, Rauber and Rünger, 2010]

A parallel program can be modeled as a task dependency Directed Acyclic Graph (DAG), defined as $G = (V, E)$, where V is a set of nodes representing parallel tasks, and E is a set of directed edges denoting dependencies between tasks [Blumofe and Leiserson, 1993, El-Rewini and Abd-El-Barr, 2005, Kessler and Keller, 2007, Kumar, 2002, Kwok and Ahmad, 1999, Narlikar and Blelloch, 1998]. Task dependencies impose ordering constraints upon the schedule and in-

troduce communication and synchronization overhead [Belikov et al., 2013, Kessler and Keller, 2007, Rauber and Rünger, 2010]. If a set of interdependent tasks are scheduled to execute in parallel, they are bound to interact through the communication subsystem to exchange data or synchronize. Contrariwise, if the same interdependent tasks execute sequentially, on the same processor, communication and synchronization overhead are reduced, but so is the ability to maintain an even distribution of work.

The key to maximum resource utilization lies in maintaining a uniform allocation of tasks to processors and ultimately keeping all units busy throughout execution. Unfortunately, determining an optimal allocation is a bin-packing problem known to be $\mathcal{NP}$*-hard* [Garey and Johnson, 1979]. Practical solutions rely on polynomial-time heuristics that can schedule computations to satisfy pre-established performance metrics. [Belikov et al., 2013, Brucker, 2007, Kwok and Ahmad, 1999, Quinn, 2004, Rauber and Rünger, 2010]

A *static* assignment, at configuration time, is best suited for loosely-coupled configurations dealing with computation-intensive applications, where the degree of parallelism rarely changes at runtime and the execution behavior is accurately known. Our work focuses on the more complex case of multithreaded algorithms, appropriate for tightly-coupled architectures such as multicore processors. The use of parallel programming artifacts inside control blocks, like conditional branches and loop iterations, results in a type of input-dependent and non-deterministic parallelism characterized by irregular execution [El-Rewini and Abd-El-Barr, 2005]. The unpredictable nature of this model requires that scheduling and load-balancing decisions be performed at runtime, in reaction to dynamic changes in the workload. *rauber11, quinn04, belikov13, foster95*

The quality of a *dynamic* scheduler is measured both in terms of the produced schedule and the overhead of producing it. In addition to the time and space expended in computing the schedule, special thought must be given to the costs and unpredictable effects of migrating tasks between processors as part of a dynamic load-balancing scheme [Belikov et al., 2013, Quinn, 2004]. Migrations also contribute to sources of inefficiency contingent on good locality of data, such as IPC and cache invalidation.

In section 2.2.3, we have also discussed how fine-grained algorithms can incur overly large operational and memory costs, if the inherent concurrency is exploited to the fullest. Extremely fine granularity in task-parallel applications can be efficiently handled using dynamic scheduling techniques that adjust the volume of parallelism to the characteristics of the parallel configuration and avoid excessive active parallelism [Blumofe and Leiserson, 1993, Narlikar and Blelloch, 1998].

2.4.1 Work-Sharing

Relevant work in dynamic scheduling of task-parallel applications has been proposed in Rudolph et al. [1991], with two techniques later termed *work-sharing*.

The simplest form of work-sharing uses a global *workpile*[11], from which tasks are assigned

[11] A queue of tasks.

to idle processors in First-In First-Out (FIFO) order. This makes for an optimal[12] and transparent load-balancing stategy, which is conceptually easy to implement, but does not preserve locality of data. As a purely centralized aproach, it also tends to scale poorly with an increasing number of processors as the global workpile becomes a bottleneck.

In an attempt to improve data locality and scalability, the decentralized variant organizes executable tasks in local workpiles that are privately owned and operated by each processor, and follows the principle of continuously moving tasks between workpiles to maintain a equitable distribution of work. Newly created tasks are inserted onto the local workpile and remain there until completion, unless they are forced to migrate by cause of the frequent redistribution of work. The load-balancing routine is initiated regularly[13], by busy processors, as they exchange tasks with another unit chosen arbitrarily to even the size of both workpiles.

Aside from resolving the scalability bottleneck, this scheme has been shown to improve data locality and produce a distribution of work that closely matches that of the centralized approach. However, the initiative and overheads of redistributing the workload fall exclusively upon busy processors, which would otherwise execute pending work. Because idle units cannot actively pull tasks from neighboring workpiles, they are bound to stay inactive for uncertain periods of time, before work is assigned by busy processors. Without a full view of the entire mapping, an arbitrary selection of load-balancing pairs only exacerbates the problem, since it cannot single out and appoint idle units as preferential recipients of work. As a result, busy processors often exchange tasks with one another, while other units remain inactive. Furthermore, and particularly in the case of dynamic and irregular parallelism, determining the size of a workpile may not be trivial.

Another fundamental flaw of this strategy lies in continuously migrating tasks so that all units attend a similar quota of the overall computation. In practice, resource utilization is not dictated by the discrepancy in size between workpiles, but rather in the amount of work that is actually performed by each processor. As a result, tasks are often migrate unnecessarily, and possibly several times, before having a chance to execute.

2.4.2 Work-Stealing

Work-Stealing (WS) is a practical and prevalent technique to schedule dynamic, fine-grained, and fully-strict[14] parallel applications. Proposed by Blumofe and Leiserson [1999], the WS scheduler is a proven efficient scheduling algorithm with bounded time, space, and communication. With a *greedy* approach on load-balancing, idle units take the initiative to fetch work from busy processors, in a strategy that offers better efficiency and performance metrics than work-sharing.

The WS scheduler uses one *worker thread*, or *worker*, per core, to organize ready tasks in a local double-ended queue, or *deque*. The worker manages its deque like a stack, pushing and popping threads from the bottom in a Last-In First-Out (LIFO) manner.

Execution begins with all deques empty. As a master task becomes ready for execution, it is

[12] In the sense that the workload is distributed across processors as evenly as possible.
[13] An interval that is inversely proportional to the size of the local workpile.
[14] In the task dependency DAG, all data dependency edges from a thread point to its parent.

assigned to a worker which, upon finding its local deque empty, directly assigns the master task to its local processor. From the moment it begins execution, the master task can enter a parallel region at any time. As a new thread is spawned, the master task is pushed to the bottom of the deque, and the new thread is given processor time straight away. Thus, since threads usually share some data with their parents, it is very likely that a part of the shared is still in cache [Acar et al., 2000].

When a thread completes, the worker looks for executable tasks at the bottom of its deque. If the deque has pending work, the bottom-most thread is popped and selected for execution. In the likely case that the bottom-most thread is a parent or sibling of the previous one, this maneuver also promotes data locality.

So far, all operations performed by the workers are completely independent, as threads remain local to increase scheduling granularity, exploit data locality, and provide low scheduling contention [Narlikar, 1999]. Synchronization between workers occurs when a local deque is found empty. Upon failing to fetch local work, the idle worker, or *thief*, launches the load-balancing procedure to make an attempt at stealing threads from another randomly chosen worker, termed *victim*. If the victim has pending work, the top-most thread of its deque is popped, migrated, and assigned to the thief. Otherwise, the WS procedure runs again, in search for eligible victims with pending work.

In the example of Figure 2.3, $Task_1$ was attributed to P1 and entered a parallel region as it spawned $Thread_{1,1}$. The newly spawned $Thread_{1,1}$ took the processor, and $Task_1$ was enqueued at the bottom of the local deque. Upon finding its local deque empty, P2 stole $Task_1$ from the top of Deque A. $Task_2$ was assigned to P3 and remained on a sequential region. P4 stayed idle, waiting for work to be assigned or for a WS opportunity.

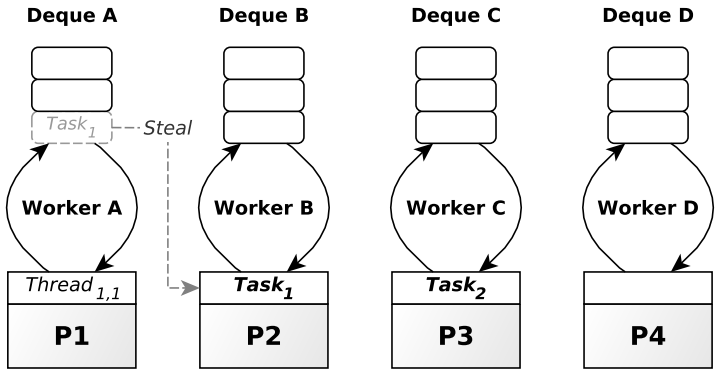

Figure 2.3: Work-stealing scheduling on a 4-processor system

Under WS, load-balancing costs are imposed upon the idle worker which would otherwise waste usable processing cycles. Locality is favored once again by stealing in a FIFO order, since top threads are the most likely to (i) generate future parallelism [Frigo et al., 1998], and (ii) have had their cache data replaced[15] by the time they are stolen. In a WS deque, all manipulations run in constant $O(1)$ time regardless of the number of threads. Since thieves and victims operate on

[15]Because top-most threads have been waiting the longest for processor time.

opposite ends of a deque, non-blocking deques can be implemented to minimize synchronization costs [Arora et al., 1998, Chase and Lev, 2005, Hendler et al., 2006].

If T_∞ denotes the minimum execution time of a fully-strict computation on an infinite number of processors, T_1 its minimum serial execution time, and scheduling overhead is ignored, Blumofe and Leiserson [1993] proves that the expected time T_p to execute the computation on m processors is given by Equation 2.2.

$$T_p \leq \frac{T_1}{m} + T_\infty. \tag{2.2}$$

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 Equation 2.3, 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.

$$S_p \leq \frac{S_1}{m}, \tag{2.3}$$

One approach to schedule parallel applications using WS, is to include the calls to a user-space runtime library that manages the threads explicitly in the application. This technique affects productivity, since it requires that the programmer be fully aware of the runtime library and the details of scheduler. Hence, WS schedulers generally resort to an alternative approach, where the parallelism is expressed at a higher-level of abstraction using parallel constructs in a programming language. This code is then transformed, by the compiler, into an equivalent version of the program with appropriate calls to WS runtime library.

Existing user-level WS schedulers are not effective, suffering from both system throughput and fairness problems, in the increasingly common setting where multiple applications share a single multi-core machine. To switch from the current support of user-space runtime mechanisms and benefit from native Operating System (OS) support, WS schedulers can also be implemented at the kernel level, as long as the OS provides basic threading and concurrency control features. [Ding et al., 2012]

2.5 Summary

Chapter 3

Real-Time Systems

"Fast is fine, but accuracy is everything."
— Xenophon

Real-time computing refers to a particular area of computer science where executions are subject to specific constraints on their response-times [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011, Laplante and Ovaska, 2011, Mall, 2009, Stankovic, 1988].

Examples of time-sensitive applications can be found across many domains of modern technology. While the monitoring and control functions of an artificial pacemaker must run at precise instants to ensure the well-being of the patient, an automotive *airbag* system must be deployed within milliseconds after collision, a multimedia decoder is set to process frames at a constant rate, and so on. All of these applications have one thing in common, in that their correctness does not solely depend on the programming logic, but also on their ability to produce results in a timely manner [Baruah and Goossens, 2003, Burns and Wellings, 2009, Laplante and Ovaska, 2011, Stankovic, 1988]. With human lives at stake, the first two examples are of highly critical scenarios where any delays can have catastrophic consequences and timing correctness must be guaranteed beforehand. When a single real-time task monopolizes the entire system[1] this is primarily a matter of ensuring that its execution demands can be fulfilled with the available resources. In multitasking environments, where multiple activities compete for execution, tasks must be scheduled in such way that each receive the minimum quota of processing bandwidth needed to respond in time. To provide *a priori* timeliness guarantees it is imperative that such schedule be deterministic [Stankovic, 1988].

Traditionally prominent in industrial applications, real-time support is becoming ever relevant in the realm of personal technology, as time-sensitive applications become ubiquitous, larger, and complex [Burns and Wellings, 2009, Mall, 2009]. Historically, great emphasis has been put on the simplicity of Real-Time Systems (RTSs), in an attempt to improve accuracy and reduce unpredictability to a minimum. As hardware and software specifications evolve, so does the difficulty to achieve predictable performance, and real-time research remains an intriguing area

[1] Either executing alone or at the highest priority level (before any other concurrent task).

of computer science with many open issues to be addressed.

From this point on, we will leave the diversity of multiprocessor architectures presented in Chapter 2 and limit our study to the problem of scheduling real-time applications on Symmetric Multiprocessing (SMP) multi-core chips, comprised of m tightly-coupled unit-capacity processors. We will start by defining real-time applications and analyzing their main characteristics. After a brief mention of Open Real-Time (ORT) systems, we conclude with an extensive study of real-time scheduling algorithms and techniques related to our work.

3.1 Task Model and Terminology

A real-time scheduler acts upon a group of real-time tasks concurrently requesting processor time, entitled *taskset*. We formalize the term real-time *task* as a schedulable unit of control that is subject to specific restrictions, called *deadlines*[2], on the amount of time it takes to produce results. A real-time task is a collection of successive instances of execution, or *jobs*, triggered by repetitive processing requests raised in their environment, and set to respond within a time frame bounded by the deadline. [Cottet et al., 2002, Laplante and Ovaska, 2011, Mall, 2009]

The foremost goal of a real-time scheduler is response-time determinism. Real-time schedulers must assess the *a priori* schedulability of the system to ensure that the timing demands of all concurrent applications are met. This is known as *schedulability analysis* [Baruah and Goossens, 2003, Cottet et al., 2002, Laplante and Ovaska, 2011]. On a given platform, tasks are said to be *schedulabe* if their real-time demands are guaranteed by the process scheduler, and *feasible* if there exists a solution to schedule the taskset in the same platform[3] [Nelissen, 2013]. Straightforwardly, a taskset is schedulabe or feasible if all of its tasks are schedulable or feasible, respectively. [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011, Laplante and Ovaska, 2011]

The richness of the information known in advance has a direct impact on the complexity and quality of scheduling decisions [Cottet et al., 2002]. For instance, all of the real-time schedulers studied herein employ schedulability analysis techniques based on task utilization. The *a priori* utilization of a task is easily determinable if the system is provided with enough detail on its execution needs. However, tasks may exhibit irregular or unknown execution behavior that cannot be characterized beforehand. Section 3.1.1 delves deeper into this subject, as we classify real-time tasks by their periodicity and criticality.

3.1.1 Taxonomy of Real-Time Tasks

In real-time theory, tasks that attend processing requests at equally distant instants are called *periodic* real-time tasks. Periodic tasks are common in sensor-actuator environments performing critical monitoring and control operations, but serve a wide variety of other purposes. With

[2]Other constraints may be considered Mall [2009], but we restrict our study to the more common model in which tasks are subject to response-time deadlines.

[3]Note that if a taskset is schedulable on a given platform, then it is also feasible in that same platform. However, the opposite does not necessarily hold.

precise knowledge of their execution needs, *a priori* schedulability guarantees for periodic tasks are straightforward. [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011, Laplante and Ovaska, 2011, Mall, 2009]

Sporadic tasks refer to executions that do not follow a strict periodic behavior, but for which a minimum inter-arrival time between jobs can be provided. This value can be used to derive the worst-case utilization of the task and guarantee its real-time needs in a schedulability test [Mok, 1983]. However, by reserving an excessive[4] amount of processing bandwidth for sporadic tasks, this approach tends to yield significant under-utilization of processing capacity. Advanced techniques presented in Section 3.3 are able to overcome this issue. [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011, Laplante and Ovaska, 2011]

On the other hand, *aperiodic* real-time tasks handle events that do not follow a known consistent cadency. Lacking information about the incidence of aperiodic tasks, no scheduler can provide *a priori* deadline guarantees for this type of applications. Nevertheless, they can still benefit from real-time support. [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011, Laplante and Ovaska, 2011]

3.1.1.1 Hard vs Soft Real-Time

Response-time determinism is a measure of how accurately deadlines are respected, and the most important criterion in a RTS. Determinism and efficiency are contradicting terms, as fully predictable systems use pessimistic techniques that tend to waste a great amount of the available resources. However, not all real-time computations need the same degree of determinism, and systems may occasionally overlook deadline correctness as a means of improving resource utilization.

Hard Real-Time (HRT) tasks are those in which delays are simply not acceptable. They are a specific type of critical executions, obliged to meet all their deadlines under penalty of complete system failure [Burns and Wellings, 2009, Cottet et al., 2002, Laplante and Ovaska, 2011, Liu and Layland, 1973]. In other words, HRT tasks are not feasible if the fulfillment of their deadlines cannot be guaranteed. To provide such guarantees for a certain task, the system must be informed of the minimum inter-arrival time between consecutive jobs, and provided an upper bound on the execution time of all jobs. This value is referred to as the Worst-Case Execution Time (WCET) of the task.

However, by referring to the execution time of the most prolonged job, the WCET offers a pessimistic view on the amount of time that jobs usually take to complete. Once a task is deemed schedulable, a fraction of the available processing capacity is reserved to guarantee that all jobs meet their timing constraints, and the bandwidth left available for other tasks decreases by the same amount. Scheduling a task by its WCET implies reserving an excessive amount of processing capacity, which is secluded from other executions and never fully consumed by the task [Pellizzoni and Caccamo, 2008]. This is particularly problematic in tasks of variable length, where the WCET case is rare and the average execution time is substantially lower.

[4] In comparison with the real execution requirements.

The wasteful impact of the WCET also depends on the accuracy of the estimation. Even in strictly controlled HRT environments, an accurate estimation of the WCET often not trivial. Unless the system is simple enough to derive the value through code and hardware analysis, it is usually determined through statistic response-time tests [Kopetz, 2011, Lee et al., 2007]. In short, WCET scheduling is prone to wasting processing bandwidth if (i) the estimation is not precise, or (ii) real execution times are considerably variant.

Variable workload is common in time-sensitive applications such as an MPEG decoder, which is likely to process key frames much faster than predicted frames. Predicted frames are also expected to take variable amounts of time to be decoded, and the processing time of a particularly intricate frame can have a huge impact on the WCET, which should be much larger than the mean execution time and quite unlikely to occur. Although it behaves much like a periodic real-time task, rhythmically spawning jobs in response to frame decoding requests, modeling such process as a HRT task would be far from ideal due to the wasteful nature of a WCET scheduling strategy. However, the Quality of Service (QoS) of an MPEG decoder is measured by its ability to generate frames at a constant rate, and a slight QoS degradation is generally admissible as long as a tolerable output stream is maintained. Though this type of applications can benefit from real-time support, they clearly do not require the same degree of determinism as HRT tasks.

To deal with this scenario, real-time literature defines *Soft Real-Time (SRT)* tasks as non-critical time-sensitive executions that may occasionally overlook their timing constraints [Burns and Wellings, 2009, Cottet et al., 2002, Laplante and Ovaska, 2011]. SRT tasks are allowed to overrun, potentially degrading the quality of the system (to a bounded amount), as long as any tardiness effects are isolated from other tasks[5]. To improve hardware utilization and eliminate the need for knowledge of the WCET, SRT tasks are scheduled by their mean execution times, and their feasibility is not determined by total deadline correctness, but by the desired level of QoS.

The peculiarities of HRT and SRT scheduling will be addressed in more detail in Section 3.3.

3.1.2 Task Terminology

Each real-time task τ_i generates a pontentially infinite number n of jobs, $\{j_{i,1}, j_{i,2}, ..., j_{i,n}\}$. If a minimum inter-arrival time between consecutive jobs can be provided, this value is referred to as the *period* T_i of the task. The *relative deadline* D_i sets a bound on the amount of time available for each job to respond. C_i denotes the *execution time* per job. Finally, O_i sets the *offset* for the release of the first job. A concise summary of this notation can be found in table 3.1.

Depending on the known execution details, some of these criteria may not be available for a certain task. The deadline is a mandatory attribute for our model of real-time tasks, and literature categorizes deadlines into three distinct types [Baruah and Goossens, 2003]:

- *Constrained* - The deadline of a task τ_i cannot be greater than the period ($D_i \leq T_i$).
- *Implicit* - The deadline a task τ_i is equal to the period ($D_i = T_i$).
- *Arbitrary* - The deadline may take an arbitrary value.

[5] To assure the schedulabiliy of concurrent tasks cohabiting in the system.

The period, however, is only observable for tasks with a constant or minimum inter-arrival time between consecutive jobs[6]. Any HRT task τ_h must either be periodic or sporadic, and C_h must refer to the WCET in order to assure that no job $j_{h,j}$ executes for more than C_h units of time in each period T_h. Following such guarantee, the *utilization* of τ_h is given by $U_h = \frac{C_h}{T_h}$, as a measure for the ratio of processing bandwidth assigned to τ_h. On the other hand, SRT tasks may be periodic, sporadic or aperiodic, and for a SRT task τ_s, C_s represents the mean job execution time. Since some jobs of τ_s are expected to take more than C_s units of time to execute, the *a priori* worst-case utilization value cannot be directly determined as $\frac{C_s}{T_s}$. One way to work out the utilization of a SRT task, is to impose and enforce a limit upon its execution demands. This case will be detailed in Section 3.3.

Each job $j_{i,j}$ of a task τ_i appears in the system at *release time* $r_{i,j}$, has a real *execution time* of $e_{i,j}$, a *relative deadline* $d_{i,j}$, and completes at *response-time* $f_{i,j}$. For quick reference, this notation is resumed in table 3.2.

A graphical representation for the execution of two jobs $j_{i,j}$ and $j_{i,j+1}$ of a generic periodic task τ_i, is also presented in Figure

The relative deadline $d_{i,j}$ of a job $j_{i,j}$ can be derived from the absolute deadline D_i as $d_{i,j} = r_{i,j} + D_i$. It sets an instant of time before which $j_{i,j}$ is expected to respond. Since a HRT task τ_h is obliged to meet all deadlines, then $f_{h,k} \leq d_{h,k} \forall k \in \{1, .., \infty\}$ immediately follows.

A SRT job $j_{s,j}$ is allowed to execute past the deadline $d_{s,j}$, and the term *tardiness* refers to the amount of time by which the deadline is missed. In this case, at any time instant t, if $d_{s,j} < t \leq f_{s,j}$, then $j_{s,j}$ is said to be *tardy*, or in *overrun*, and the tardiness is given by $t - d_{s,j}$.

Notation	Interpretation	Constraint / Definition
τ_i	Task i	
C_i	Execution time of τ_i	$C_i > 0$
T_i	Period of τ_i	$P_i > C_i$
D_i	Relative deadline of τ_i	$D_i \geq C_i$
O_i	Offset of τ_i	$O_i \geq 0$
U_i	Utilization of τ_i	
j_i	Arbitrary job of τ_i	
$j_{i,j}$	Job i, j of τ_i	$j \geq 1$

Table 3.1: Task notation summary.

Notation	Interpretation	Constraint / Definition
$j_{i,j}$	Job j of τ_i	
$r_{i,j}$	Release time of $j_{i,j}$	$r_{i,j} \geq r_{i,j-1} + T_i$
$d_{i,j}$	Absolute deadline of $j_{i,j}$	$d_{i,j} = r_{i,j} + D_i$
$f_{i,j}$	Response time of $j_{i,j}$	$f_{i,j} \geq r_{i,j}$

Table 3.2: Job notation summary.

3.2 Open Real-Time Systems

Unlike classical *closed systems*, where every single real-time parameter is known beforehand, the composition and configuration of *Open Real-Time (ORT)* systems is flexible and dynamic. In an

[6]Periodic or sporadic tasks.

ORT system, HRT, SRT, and non real-time applications, concurrently contend for execution in a shared processor, and are allowed to enter or leave the system at any time. [Deng and Liu, 1997]

ORT systems are of extreme economic and technological importance in the modern market. In avionics, for example, where the size, weight, power, and cost of each hardware component are important, the ability to integrate different independently-developed applications in a single machine is immensely beneficial. Also, as an increasing number of users run both real-time and traditional desktop applications in the same computer, the need to provide real-time support in General-Purpose Operating Systems (GPOSs) gains a whole new relevance.

Our vision of an ORT environment is one which takes a two-level scheduling approach to handle real-time and non real-time tasks separately. Real-time tasks are prioritized over general-purpose applications, and scheduled through an algorithm able to satisfy their timing requirements, while other applications are scheduled in the background. As GPOSs supply powerful solutions for non real-time executions, we direct our attention towards dynamic real-time scheduling and the overall problematic of satisfying real-time demands in GPOSs.

A full schedulability analysis in an ORT environment can be complex, and notoriously intricate in the case of dynamic multithreaded applications, executions of variable length, or due to sources of unpredictable overhead [Deng and Liu, 1997]. Despite the lack of predictability, the system must employ means to simultaneously guarantee temporal correctness and reasonable throughput. Fulfilling these goals in a general and open environment is not straightforward, and continues to pose great challenges to the scientific community.

Promising approaches, presented in Section 3.3, have arisen to provide the needed scheduling support for ORT systems. Achieving predictable performance typically entails the use of *temporal isolation* and *admission control* techniques. Temporal isolation is employed to ensure that the temporal correctness of task depends only on its own consumption of processing resources, and that erroneous tasks do not jeopardize the correctness of the entire system. Admission control imposes restrictions upon the acceptance of dynamic real-time requests, to accurately assure that the arrival of a new task does not overload the available processing capacity and disrupt the timing correctness of other concurrent applications.

3.3 Real-Time Scheduling Theory

In multitasking systems, many performance indexes depend on how the finite resource of Central Processing Unit (CPU) time is apportioned among concurrent executions. Computing science refers to the problem of distributing processing time among tasks as *CPU scheduling*, or *scheduling*. In a computing platform, the subsystem accountable for ordering the use of the CPU is known as *process scheduler*, henceforth simply called *scheduler*. [Bovet and Cesati, 2005, El-Rewini and Abd-El-Barr, 2005, Laplante and Ovaska, 2011, Love, 2010]

The set of rules implemented by a scheduler, defining the logic by which tasks are selected for execution, is called a *scheduling policy* [Bovet and Cesati, 2005, Love, 2010]. Schedulers are evaluated by their complexity and by the quality of the produced schedule, thus they must decide quickly and towards the optimization of specific performance criteria such as fairness, throughput,

predictability, *et cetera* [El-Rewini and Abd-El-Barr, 2005]. As some of these indexes may be conflicting, they must be in tune with the overall goals of the system. For instance, while GPOSs seem to favor interactivity and resource utilization through a fair distribution of processor shares, RTSs require a completely distinct strategy to guarantee that tasks execute and respond in a timely manner, even if the CPU is not utilized to the fullest.

Scheduling theory has long been a theme of great debate among the real-time community. Over the years, many developments have relied on simple, predictable, and well characterized applications, to establish highly efficient schedules at configuration time. While such model has some obvious strong points, it is also very limiting. We study the more intricate case of tasks with unpredictable or variable execution requirements that can enter or leave the system at random, calling for dynamic scheduling mechanisms capable of reacting to load changes at runtime. Unfortunately, these methodologies are prone to inconsistent runtime costs[7], which threaten the deterministic guarantees expected from a RTS. [Burns, 1991, Kopetz, 2011, Laplante and Ovaska, 2011, Lee et al., 2007]

A common classification of real-time schedulers is based upon whether or not task priorities remain constant throughout execution. *Static-priority*, or *Fixed-Priority (FP)* schedulers, consider that the priority of each task is inherited by all of its jobs and remains constant throughout execution. *Dynamic-priority*, or Job-Level Fixed-Priority (JLFP), scheduling approaches abide by a more flexible strategy, in which different jobs of the same task are assigned different priorities at runtime. In most dynamic-priority schedulers, including the ones studied in this thesis, the priority of a job is defined by its absolute deadline. [Baruah and Goossens, 2003, Cottet et al., 2002, Kopetz, 2011, Laplante and Ovaska, 2011, Lee et al., 2007, Mall, 2009]

From another perspective, multitasking real-time schedulers can be further classified as *preemptive*, *non-preemptive*, or *cooperative* [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011], according to the following definitions:

- Preemptive - Executing jobs can be swapped for higher priority jobs at any time instant.
- Cooperative - Specific preemptable sections exist within the execution of a job.[8]
- Non-preemptive - Preemptions are not allowed and jobs execute to completion.

To guarantee real-time demands, schedulers implicitly assign a fraction of the available processing bandwidth to each task. Conventionally, a uniprocessor platform has a total bandwidth of 1. The bandwidth of an SMP chip comprised of m identical processors is given by the sum of the bandwidth in each individual unit, and thus equal to m. Evaluating the *a priori* schedulability of a taskset, is a matter of analyzing whether its total utilization fits within the schedulable utilization of the algorithm on a given platform. In real-time literature, this is known as a *schedulability test*. Following the same principle, tests that assess the viability of dynamic changes in the schedule at runtime, are more commonly referred to as *admission tests* or *acceptance tests*. [Burns and Wellings, 2009, Cottet et al., 2002, Kopetz, 2011, Lee et al., 2007, Mall, 2009]

[7] Caused by factors such as resource contention, context switching, task migration, among others.

[8] For example, a scheduler that preempts tasks at a constant rate.

A real-time scheduler is said to be *optimal*, with respect to a given platform and task model, if it can guarantee the schedulability of every taskset that is generated according to the model and feasible in the same platform. In other words, an optimal scheduling algorithm can always find a feasible schedule whenever it exists. [Carpenter et al., 2004, Cottet et al., 2002, Kopetz, 2011, Mall, 2009]

Theoretical proposals idealize simplistic hardware and software models to prove mathematical correctness and efficiency, often assuming that: [Laplante and Ovaska, 2011]

- Tasks are independent and do not share resources (other than the processor).
- The timing behavior is deterministic and events occur exactly when they are supposed to.
- Executions do not block except in between jobs. At any time, they are either executing or awaiting execution.
- There is no runtime overhead inherent to scheduling decisions, such as task migration and context switching.

Although necessary to legitimate a proposal and devise mathematical schedulability tests, these assumptions can be problematic in the practical case. Many algorithms are yet to be tested in real settings, where many sources of unpredictability and overhead tamper with the optimistic theoretical observations. From experience, when attempting to implement a scheduling algorithm it is crucial to make correct structural choices, and employ the right programming practices, to mitigate the effects of unavoidable overheads disregarded in the proposal.

3.3.1 Uniprocessor Scheduling

Foundational real-time scheduling research dates back to the late 1960's, in the ambit of the first manned mission to the moon. [Liu, 1969]

The highly influential work of Liu and Layland [1973] formalizes two preemptive FP scheduling algorithms for the periodic implicit-deadline ($T_i = D_i$) HRT task model: Rate-Monotonic (RM) and Earliest Deadline First (EDF). The RM scheduler assigns the higher priority to tasks with lower period T_i. At each time, the active task with the lowest value of T_i is scheduled for execution.

Figure depicts a RM scheduling scheme for a set of two tasks ...

As proved by Liu and Layland [1973], a taskset of n tasks, with total utilization of $U_{sum}(\tau) = \sum_i^n U_i$, can be scheduled by RM if the sufficient test in Equation 3.1 holds.

$$U_{sum}(\tau) \leq n(2^{\frac{1}{n}} - 1) \tag{3.1}$$

This shows that RM is not optimal, since it cannot exploit full processing capacity with all tasksets. In fact, and besides the total utilizaition of the taskset, the schedulability test depends on the number n of tasks competing for processor time, and as the number of concurrent tasks approaches infinity, it follows that $\lim_{n\to\infty}\left(n(2^{\frac{1}{n}} - 1)\right) = ln(2) \approx 0,693$. Therefore, as a

sufficient condition, the RM schedulability of any taskset can be guaranteed *a priori* if its total utilization does not exceed approximately $69,3\%$ of the available bandwidth.

Derived from RM, the preemptive Dealine-Monotonic (DM) scheduler was proposed by Leung and Whitehead [1982], to schedule tasks with constrained deadlines ($D_i \leq T_i$) more efficiently. DM schedules HRT tasks by their relative deadline D_i, assigning higher priority to tasks with lower value of D_i. Figure exemplifies the scheduling scheme for two tasks under DM.

Under an implicit-deadline model, DM scheduling is analogous to RM, thus efficiency improvements are only observable for cases where $D_i \leq T_i$.

3.3.1.1 Earliest Deadline First (EDF)

Also introduced by Liu and Layland [1973], EDF is the most studied dynamic-priority scheduling algorithm. It follows the simple idea of scheduling tasks by their urgency. In other words, the closer a task is to its relative deadline, the higher its priority, and the sooner it will be scheduled for execution.

EDF is a proven optimal scheduling algorithm for HRT and SRT computations in uniprocessor systems. The test in Equation 3.2 shows that EDF can exploit full capacity for HRT tasksets τ with a total utilization of $U_{sum}(\tau)$. On the other hand, if $U_{sum}(\tau) < 1$, the resulting unused capacity $1 - U_{sum}(\tau)$ can be used to schedule SRT executions with bounded tardiness. However, if jobs of a SRT task τ_i take more than C_i units to execute, the overused capacity can be taken from other computations, and compromise the bandwidth guarantees for critical HRT tasks.

$$U_{sum}(\tau) = \sum_{i}^{n} U_i \leq 1 \tag{3.2}$$

Equation 3.2 shows a major improvement in comparison with RM 3.1, but towards an efficient implementation of EDF, a number of complications need to be addressed and resolved. For instance, Operating Systems (OSs) make use of priority queues, arrays, or bitmaps, to perform quick scheduling decisions. Maintaining the organization of such structures requires frequent and costly computations, which introduce runtime overhead. [Short, 2010]

Many argumentative statements claim that RM offers significant advantages over EDF. RM is said to be easier to analyze, less prone to runtime overhead, more predictable, and more efficient under transient overload. These allegations were discredited with a sufficient level of confidence in Buttazzo [2005]. Though they may be valid in some cases, and should not be neglected, such affirmations offer no reason for why EDF scheduling in Real-Time Operating Systems (RTOSs) should be disregarded.

3.3.1.2 Constant Bandwidth Server (CBS)

Using average execution times to schedule SRT tasks may have undesirable consequences when jobs take more than requested value to execute, and for these cases, the system must be able to isolate any overrun effects from other concurrent executions. Reservation-based methodologies

introduce the possibility to assign a fraction of the available bandwidth to executions, and ensure that eventual tardiness effects do not propagate to other tasks

In their proposal of the Constant Bandwidth Server (CBS) scheduler, Abeni and Buttazzo [1998] improved the technique presented by Mercer et al. [1994] for the RM policy, to handle SRT requests with a variable or unknown execution behavior through EDF. To avoid unpredictable delays in HRT tasks, SRT tasks execute in the context of a *bandwidth server* that reserves a fraction of the available processing capacity and ensures that SRT tasks never demand more than the reserved bandwidth. Although not mandatory, to ease discussion we consider a one-to-one[9] relationship between bandwidth servers and SRT tasks.

A bandwidth server S_i is characterized by a couple (Q_i, T_i) indicating that a SRT task can execute for, at most, Q_i units of time in each period T_i. The j^{th} job of a SRT task τ_i, served by S_i, is released at time $r_{i,j}$, with a budget $c_{i,k} = Q_i$, and a deadline $d_{i,k} = r_{i,j} + T_i$. As long as τ_i is executing, the budget $c_{i,k}$ is decremented at each instant by the same amount. If the budget is exhausted ($c_{i,k} = 0$) at time $t < f_{i,j}$, then a new server instance $k + 1$ is created, as $c_{i,k+1}$ is recharged to Q_i and the deadline is postponed by T_i, as $d_{i,k+1} = d_{i,k} + T_i$. Postponing the deadline also decreases the priority[10] of τ_i, forcing it to yield the processor to higher priority tasks. As a result, (i) the reserved bandwidth is not exceeded, and (ii) tardiness effects are isolated from the system and only affect the served task.

Unfortunately, if a job $j_{i,j}$ completes early with $c_{i,k} > 0$, the remainder of $c_{i,k}$ cannot be used by other tasks. Unless a subsequent job $j_{i,j+1}$ is released at time $r_{i,j+1} < d_{i,k}$ to consume $c_{i,k}$, the leftover capacity is wasted and left unused. Therefore, the performance of CBS highly depends on a precise allocation of processor shares to avoid underusing computational bandwidth.

A given server S_i has a bandwidth (or utilization) of $U_i^{(s)} = \frac{Q_i}{T_i}$, and the total utilization for soft real-time tasks is given by the sum of the bandwidth reserved by all n servers, as shown in Equation 3.3.

$$U_{sum}^{(s)} = \sum_{i}^{n} U_i^{(s)} \tag{3.3}$$

Therefore, and since it is guaranteed that SRT tasks do not consume more than $U_{sum}^{(s)}$, the remaining bandwidth can be used to guarantee the feasibility of HRT tasks, as if they were executing alone in a processor of capacity equal to $1 - U_{sum}^{(s)}$. Being $U_{sum}^{(h)}$ the conjoint utilization of all HRT tasks, the admission test for a taskset τ under the CBS scheduler is presented in Equation 3.4.

$$U_{sum}(\tau) = U_{sum}^{(s)} + U_{sum}^{(h)} \leq 1 \tag{3.4}$$

[9] Each server manages exactly one task, and each task is bound to exactly one server

[10] The priority of a task, under EDF, is inversely proportional to the value of its relative deadline.

3.3.1.3 Capacity-Sharing (CASH)

To overcome the wasteful reservation strategy of CBS, several bandwidth reservation schemes allow other tasks to reuse the residual capacity originated by early completion of served jobs. [Caccamo et al., 2000, 2005, Lin and Brandt, 2005, Lipari and Baruah, 2000, Marzario et al., 2004]

In the Capacity-Sharing (CASH) scheduler [Caccamo et al., 2000], each task τ_i (hard or soft) is bound to a dedicated server $S_i = (Q_i, T_i)$ scheduled by EDF. HRT tasks are scheduled by their WCET, and SRT tasks by their mean expected execution time and period. A server S_i is said to be *active* if there are pending jobs of τ_i to be served, and *idle* otherwise. At each instant, the remaining execution capacity $0 \leq c_{i,k} \leq Q_i$, and the relative deadline $d_{i,k}$, are associated to S_i.

If a new job $j_{i,j}$ is created at time $r_{i,j}$ and the server is idle, a new server deadline is generated as $d_{i,k} = max(r_{i,j}, d_{i,k-1}) + T_i$, and $c_{i,k}$ is recharged to Q_i. If the server is active at time $r_{i,j}$, the new job is enqueued in a work queue of S_i and awaits execution with the current deadline $d_{i,k}$.

CASH considers a system-wide queue of residual capacities, so that when a server S_i becomes idle at time t with $c_{i,k} > 0$, a new residual capacity is inserted in the *CASH queue* with a capacity $c_q = c_{i,k}$ and deadline $d_q = d_{i,k}$. For the sake of correctness, as the capacity of S_i is being donated, $c_{i,k}$ is set to 0.

When a job $j_{i,j}$ is scheduled for execution, if there is a residual capacity q in the CASH queue, such that $d_q \leq d_{i,k}$, the residual capacity is removed from the queue, and $j_{i,j}$ executes with $d_{i,j} = d_q$, consuming c_q instead of its dedicated capacity $c_{i,k}$. If the CASH queue is found empty, or if no residual capacity meets the requirement $d_q \leq d_{i,k}$, then $j_{i,j}$ executes normally with dedicated capacity $c_{i,k}$ and deadline $d_{i,k}$. At each time t, any residual capacity q that has been exhausted or expired ($c_q = 0 \vee d_q \leq t$), is removed from the CASH queue.

Whenever the capacity $c_{i,k}$ of an active server S_i is exhausted ($c_{i,k} = 0$) a new instance of S_i begins, as the budget c_i is recharged to Q_i and d_i is postponed ($c_{i,k+1} = Q_i$ and $d_{i,k+1} = d_{i,k} + T_i$).

Straightforwardly, the total utilization is given by the sum of all n individual reserves assigned to each dedicated server, and the admission test for the CASH scheduler is given by Equation 3.5.

$$U_{sum}(\tau) = \sum_{i}^{n} U_i \leq 1 \tag{3.5}$$

In Caccamo et al. [2000] CASH was proven correct and more efficient than CBS to schedule tasks of large execution variance, with a vestigial increase in runtime overhead. Its adaptation to the multiprocessor paradigm, presented in 3.3.2.4, is particularly relevant to our work.

3.3.2 Sequential Multiprocessor Scheduling

As observed by Liu [1969], uniprocessor real-time scheduling theory is not directly applicable to multiprocessors. With several processing units, the scheduling problem becomes a multi-dimensional one, as the scheduler must decide not only when, but also where to execute each task. Multiprocessor real-time schedulers need acknowledge multiple processors, eligible to ex-

ecute a set of tasks, and devise a correct allocation of tasks to processors in terms of timing correctness and throughput.

Unfortunately, real-time scheduling upon multiprocessors is still a rather complex and immature area of computer science, where many problems prevail. Sahni [1977] shows that, unlike the class of algorithms studied so far, for serial computers, no JLFP scheduler can utilize full processing capacity in a multiprocessor platform. More recently, Fisher et al. [2010] proved that it is impossible to design an optimal online multiprocessor scheduler for the sporadic task model[11].

As noted in Chapter 2, the multiprocessor paradigm also brings about many sources of latency and unpredictability[12] that require careful analysis and trade-offs in real-time scheduling [Carpenter et al., 2004].

Furthermore, multiprocessors are subject to well-known anomalies, in which changes that appear beneficial to the system[13] can make a previously schedulable taskset become unfeasible [Andersson and Jonsson, 2002, Graham, 1976].

3.3.2.1 Global, Partitioned, and Semi-Partitioned Scheduling

Real-time schedulers are often classified in literature as global, partitioned, or semi-partitioned.

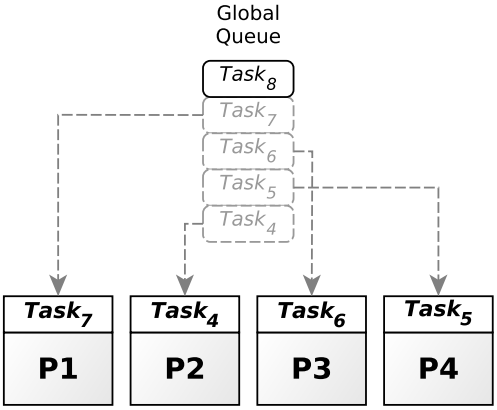

Figure 3.1: Global scheduling

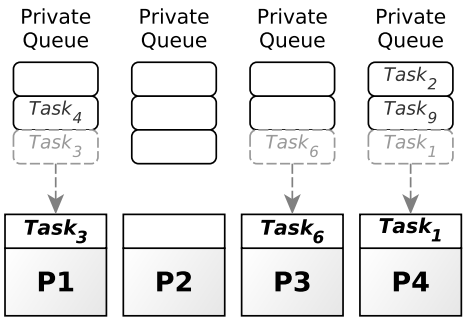

Figure 3.2: Partitioned scheduling

In *global* scheduling, tasks are selected for execution from a single system-wide priority queue. With a full view of the entire system, load-balancing is immediate and scheduling decisions are optimal, as the scheduler (Figure 3.1) need only look at the global priority scheme to keep the highest priority tasks executing on the available processing units at all times.

Global schedulers are known to offer better throughput than partitioned schedulers for implicit-deadline tasks [Kleinrock, 1976]. In fact, the class of rate-based[14] *Proportionate Fair (Pfair)* schedulers [Baruah et al., 1996] provide the only known optimal methodology to schedule periodic HRT tasks on multiprocessors [Carpenter et al., 2004].

[11] As well as more general models

[12] Migration overheads, cache invalidation, resource contention, among others intrinsic to multiprocessors.

[13] Such as reducing the utilization of a task, adding a processor, *et cetera*.

[14] Schedulers that operate at steady pre-computed instants of time.

Global schedulers are *work-conserving*, *i.e.* they migrate tasks between processors to avoid processor idling and maximize resource utilization. However, a number of factors ignored in global scheduling theory, from non-deterministic resource contention, to implementation complexity, poor scalability, and cache invalidation, often render implementations impractical [Baker, 2005a]. Dhall [1977] also noticed how a global RM or EDF scheme operating on a multiprocessor host may cause deadline misses, even with tasksets requesting less than the available capacity.

These observations had a negative influence upon the real-time community for many years. Global schedulers only regained popularity when it was realized that the Dhall effect is not exactly a problem of global scheduling, but rather of sequential tasks with high utilization [Phillips et al., 1997]. Regardless, their practicality remains controversial [Baker, 2005a].

Partitioned schedulers (Figure 3.2) perform a static and permanent allocation of tasks to private processor queues at configuration time. Here, the online scheduling problem is transformed into a series of uniprocessor ones [Carpenter et al., 2004], as each processor is scheduled individually according to a certain policy[15].

A number of advantages over global approaches can be appointed for partitioned schemes, as they allow for a thorough schedulability analysis, improve data locality, isolate overrun effects across processors, and eliminate the scalability bottleneck of concurrent access to a shared queue. Nevertheless, as an optimal *a priori* distribution of work is attempted, the $\mathcal{NP}$-hard *bin-packing* problem arises [Garey and Johnson, 1979], requiring additional heuristics to achieve a quick and, at best, satisfying allocation. The most common techniques 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 with the least amount of resources after allocations.

Unable to migrate tasks between processors, partitioned schedulers may leave some units overloaded while others remain idle (*non-work-conserving*). For the same reason, there may exist tasksets that are feasible on a multiprocessor platform, but not schedulable under a fully partitioned scheme on the same platform.

Semi-partitioned or *hybrid* schedulers either keep tasks on private queues shared by several processors (*clustered* scheduling), migrate tasks between per-processor queues throughout execution, or both. While semi-partitioned schemes look to take out the best of both approaches, they raise other sources of inefficiency when compared with pure global or partitioned schemes. Semi-partitioned schedulers can be finely tuned to suit a particular purpose. For instance, in order to to simultaneously eliminate the contention problem of a global scheme and maintain good resource utilization, tasks may be kept in processor-specific queues but migrated frequently to achieve some degree of load balance. However, at the same time as contention for a single priority queue disappears, concurrency on all queues must be managed explicitly, and the system incurs migration overhead, which is especially problematic in RTSs as a source of unpredictability. To enhance determinism, for example, one might cut down migrations by allowing tasks to move only at job boundaries, which would imply a less efficient load-balancing strategy.

Partitioned schedulers naturally match closed HRT environments, where determinism is crucial, most details of execution are known, and a static allocation is viable. Global schedulers are

[15] Different algorithms may be configured for each processing unit.

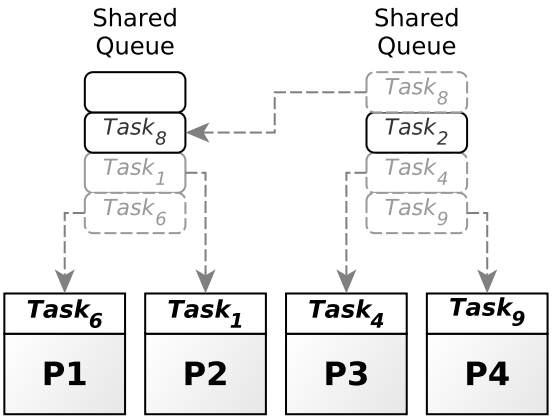

Figure 3.3: Clustered scheduling with task migration

comprehensibly better suited for open environments handling dynamic and unpredictable workloads, as considered in this thesis. [Lelli et al., 2011]

3.3.2.2 Global EDF (G-EDF)

Global Earliest Deadline First (G-EDF) is the homologous to EDF for multiprocessor platforms under a global scheduling approach. It foresees the existence of a single shared queue, from which the highest priority tasks are selected for execution on the m available processing units.

In the HRT case, G-EDF is vulnerable to severe capacity loss due to the Dhall effect, resulting in a total utilization bounded by Equation 3.6 for periodic task sets [Andersson et al., 2001], which is in effect for all multiprocessor JLFP schedulers.

$$\frac{m+1}{2} \tag{3.6}$$

G-EDF guarantees bounded tardiness for any sporadic SRT taskset τ with utilization $U_{sum}(\tau) \leq m$ [Devi and Anderson, 2008], but as its predecessor EDF, lacks bandwidth enforcement and isolation features to control the effects of SRT overruns.

Although the theoretical worst-case performance of G-EDF, in a sequential HRT context, cannot be higher than that of Equation 3.6, a number of schedulability tests guarantee a greater bound in low-utilization[16] cases. The first was introduced by Goossens et al. [2003], showing that a periodic implicit-deadline taskset τ, with total utilization $U_{sum}(\tau) = \sum_i^n u_i$ is schedulable by G-EDF on m processors if the tight bound in Equation 3.7 is met. $U_{max} = max_{1 \leq i \leq n}(u_i)$ represents the highest utilization among all tasks of τ.

$$U_{sum}(\tau) \leq m - (m-1)U_{max} \tag{3.7}$$

Several improvements of 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 tasks of utilization greater than a threshold ζ,

[16] When U_{max} is considerably less than one.

and schedules the remaining tasks according to the standard EDF policy. By setting ζ to $\frac{m}{2m-1}$, the schedulability test is given by Equation 3.8.

$$U_{sum}(\tau) \leq \frac{m^2}{2m-1} \tag{3.8}$$

[Goossens et al., 2003] also propose an algorithm that assigns highest priority to the k tasks with highest utilization. This approach, named EDF$^{(k)}$, sets the sufficient schedulability condition presented in Equation 3.9, where U_k represents the lowest utilization in the set of k tasks with the highest utilization.

$$(k-1) + \left\lceil \frac{U_{sum}(\tau) - U_k}{1 - U_k} \right\rceil < m \tag{3.9}$$

EDF-US[ζ] and EDF$^{(k)}$ were examined by Baker [2005b], showing that the optimal threshold ζ for EDF-US[ζ] is $1/2$, since it results into the maximum possible bound for this class of scheduling algorithms (see Equation 3.6):

$$U_{sum}(\tau) \leq \frac{m+1}{2} \tag{3.10}$$

With regard to EDF$^{(k)}$, Baker [2005b] shows that there exists a minimum value of k, represented as k_{min}, for which the worst-case schedulable utilization in Equation 3.10 also holds. Nevertheless, EDF$^{(k_{min})}$ outperforms EDF-US[$1/2$], concerning the number of schedulable task sets.

Baruah [2004] later proposed fpEDF, which assigns highest priority to the $m-1$ heavier[17] tasks with utilization greater than $1/2$. fpEDF guarantees the schedulability of any taskset satisfying Equation 3.11.

$$U_{sum}(\tau) \leq max\left(m - (m-1)U_{max}(\tau), \frac{m}{2} + U_{max}(\tau)\right) \tag{3.11}$$

3.3.2.3 M-CBS

The bandwidth server abstraction was extended to the multiprocessor scenario in Baruah et al. [2002], as a part of the M-CBS scheduler, to provide bandwidth reservation and isolation for SRT executions. Designed over G-EDF, it guarantees the schedulability of a set of n servers, on m identical processors, as long as their total utilization $U_{sum}(\tau) = \sum_i^n U_i$ satisfies Equation 3.12.

$$U_{sum}(\tau) \leq \frac{m+1}{2} \tag{3.12}$$

Though asymptotically optimal for JLFP executions, M-CBS can severely underuse processing bandwidth, like its uniprocessor equivalent, when SRT executions exhibit large variance.

[17] Tasks with greater utilization.

3.3.2.4 M-CASH

Succeeding the proposal of Baruah et al. [2002], the CASH mechanism was adjusted to multiprocessor platforms in the description of the M-CASH scheduler [Pellizzoni and Caccamo, 2008].

With the understanding that CASH rules are not directly applicable to multiprocessors, Pellizzoni and Caccamo [2008] successfully adapted the CASH proposal to a G-EDF scheme, and even improved some of its original limitations. As such, M-CASH considers two types of servers: *bandwidth-sharing* servers (*BS-servers*) for aperiodic tasks, and *capacity-sharing* servers (*CS-servers*) for periodic and sporadic tasks.

A CS-server S_c follows original CASH rules. When S_c becomes idle with $c_{c,k} > 0$, a new residual capacity (c_q, d_q), set as $c_q = c_{c,k}$ and $d_q = d_{c,k}$, is inserted in the system-wide CASH queue, and $c_{c,k}$ is forced to 0. In CASH, only one server is allowed to take the residual capacity, but due to multiprocessor anomalies, imposing the same rule in the multiprocessor paradigm would result in a less efficient strategy. Instead, at each time instant, an M-CASH residual capacity q is shared evenly by all eligible servers[18].

BS-servers, on the other hand, do not share residual capacities. By the original CASH rules, once a server S_i generates a residual capacity, its budget $c_{i,k}$ is set to 0, and the arrival of a new job forces a deadline postponement. As a result of successive early completions, the deadline of S_i may be set to an exaggeratedly distant value[19], significantly decreasing the priority of the served task. This is even more likely in aperiodic SRT tasks with variable execution behavior, causing future jobs to be scheduled in the background and becoming overly tardy. Instead, when a job served by a BS-server S_b completes with $c_{b,k} > 0$, its deadline $d_{b,k}$ is decremented by $\frac{c_{b,k}}{U_b}$. The idea is to maintain the unused capacity of a BS-server S_b, to attend any job requests that might appear before $d_{b,k}$. Thus, instead of donating the resulting unused capacity, the time consumed by a completing aperiodic job is subtracted from the current deadline $d_{b,k}$, and the leftover capacity $c_{b,k}$ remains available for future jobs appearing before $d_{b,k}$.

Since all tasks are assigned to a bandwidth server, which is modeled as a periodic unit scheduled by G-EDF, the utilization bound of M-CASH meets that of Goossens et al. [2003], shown in Equation 3.7.

Pellizzoni and Caccamo [2008] further propose two extensions to the core M-CASH algorithm: M-CASH-AGING, and M-CASH-fp. M-CASH-AGING effectively reduces the deadline aging problem, by allowing an executing server S_i to execute freely (*i.e.* without decreasing its own capacity $c_{i,k}$) at time t, as long as $c_{i,k} < (d_{i,k} - t)U_i$, and there exist active servers in the ready queue waiting for execution. M-CASH-fp considers the use of dedicated processing units to execute servers with large utilization $Ui > 1/2$, based on the idea of fpEDF.

3.3.3 Parallel Real-Time Scheduling

All of the proposals discussed in Section 3.3.2 take on real-time scheduling in multiprocessors, but completely disregard intra-task parallel execution.

[18] Active servers with deadlines greater than d_q

[19] An effect referred to as *deadline aging* [Pellizzoni and Caccamo, 2008].

The Dhall effect (Section 3.3.2.1) is fundamentally a problem of load-imbalance, caused by sequential tasks that cannot be divided into smaller execution chunks and executed in parallel. To the attentive reader, this goes in accordance with the observations of Amdahl presented in Section 2.2.1, which shows how the performance of the system is restrained by the sequential portion of the computation.

Multi-core architectures are currently instigating a change in programming habits. Frameworks such as OpenMP [ARB], Cilk [Frigo et al., 1998], Intel's PBB [Intel Corporation], Java Fork-join Framework [Lea, 2000], Microsoft's Task Parallel Library [Microsoft Corporation], StackThreads/MP [Taura et al., 1999], among others, now provide high-level abstraction models that allow software developers to easily parcel out their programming logic into schedulable threads, which can be executed separately across the available processing units.

With the idea that highly parallel applications offer the best opportunities for parallel execution and performance enhancement, programmers are allured to specify as much parallelism as possible. The resulting applications merely point out potentially parallel regions within their logic, thus user-code expressiveness alone does not guarantee parallel execution. Since many details of execution [20] are often not known in advance, much of the actual work of assigning threads to processing units must be performed dynamically by the underlying process scheduler.

On the other hand, automatic parallelization tools are known to break down programming logic into a particularly large number of short-lived threads. This kind of irregular and fine-grained parallelism may raise overwhelming scheduling costs that suppress the gains of parallel execution, and must be attended by simple and fast scheduling mechanisms able to keep the overhead low. [Diaz et al., 2012, Dongarra et al., 2003]

Most prior work in parallel real-time scheduling [Collette et al., 2008, Kato and Ishikawa, 2009, Kwon and Chwa, 1995, Lee and Lee, 2006, Manimaran et al., 1998] assumes that the parallelism degree of jobs is known beforehand.

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.

In practice, this information is not easily discernible, and in some cases can be inherently misleading. In the proposal of Lakshmanan et al. [2010], all parallel regions within a task have the same number of threads, which cannot exceed the number of processor cores, and all threads are equal in length. In Saifullah et al. [2011], the number of threads can and vary between regions, as well as exceed the number of cores, but still requires all threads of a region have the same size.

With remarkable publications and projects, the research team at Research Centre in Real-Time Computing and Embedded Computing Systems (CISTER) have recently been addressing

[20] Such as the number of iterations in a loop, or the number of threads in a parallel region.

the problem of exploiting intra-task parallelism in real-time applications. The Real-Time Work-Stealing (RTWS) algorithm, presented next, shows how Work-Stealing (WS) logic can be combined with G-EDF to scale the performance of HRT tasksets in multiprocessors. The Parallel Capacity Sharing by Work-Stealing (p-CSWS) scheduler, which serves as the theoretical foundation of our work, uses bandwidth reservation, capacity-sharing, and WS, to schedule dynamic and irregular parallel real-time applications in an open environment. A thorough description of the p-CSWS scheduler is presented in Section 3.3.3.2.

3.3.3.1 The RTWS Scheduler

The Real-Time Work-Stealing (RTWS) scheduler, proposed by Fonseca [2012], combines G-EDF with a priority-aware Work-Stealing (WS) strategy to schedule parallel HRT tasks on SMP platforms.

RTWS considers a HRT fork/join parallel task model. Ready tasks are scheduled by their WCETs, and organized in a global queue ordered by non-decreasing absolute deadlines, from which the first m tasks are selected for execution in m processing units. Dynamically generated threads are kept in a local processor-specific priority queue of WS deques, sorted by non-decreasing order of the deadlines assigned to their current jobs[21]. This scheme is depicted in Figure .

Each processor core looks for work in its local queue, selecting the bottom-most thread in the highest priority deque for execution. If the local queue is found empty, it tries to pull a task from the global queue. In case the global queue has no pending work, a priority-aware WS operation is attempted, as the local processor looks for the highest priority deque in the system and steals the topmost thread, which is then inserted in the local processor queue and selected for execution. Contrarily to non-stolen dynamically generated threads, stolen threads preempted by higher priority tasks do not return to the local processor queue. Instead, they are placed in the global queue.

The schedulability condition for RTWS, presented in Equation 3.13, follows the observations of Goossens et al. [2003] for G-EDF scheduling of periodic HRT tasks on a platform of m unit-capacity processors.

$$U_{sum}(\tau) \leq m - (m-1)U_{max}(\tau) \tag{3.13}$$

As in the original WS proposal [Blumofe and Leiserson, 1999], the choice to keep newly created threads in the processor they are originated, greatly favors data locality. In RTWS, parallelism is only exploited when a processor becomes idle, effectively reducing runtime overhead. Additionally, it is up to the the idle processor to take the initiative of stealing threads, as well as any overhead inherent to the operation.

The algorithm in Blumofe and Leiserson [1999] does not consider task priorities, and idle workers attempt stealing operations in randomly chosen processors until an overloaded worker is found. Such an approach is suitable for the original scenario, but not in a HRT system where

[21] At any given time, each deque stores threads of a single job.

determinism is primordial. As work gets scarce, an idle processor may spend an unpredictable amount of time trying to find an overloaded worker, and though the overhead of doing so is taken by an idle unit, processing time is still lost.

Instead, RTWS introduces a Priority-Aware Stealing (PAS) mechanism, which steals threads from the highest priority deque in the system. Precise knowledge of the workload, in each individual processing unit, demands more space and additional maintenance, but offers bounded runtime overhead, which is also effectively reduced in the long term. In a sense, the WS mechanism in Fonseca [2012] tries to exploit parallelism only when it is to benefit the overall response-time of the system, prioritizing tasks with shorter absolute deadlines.

RTWS is a provably efficient scheduling algorithm for parallel HRT applications, also able to schedule SRT tasks with bounded tardiness like any G-EDF-based scheduler [Devi and Anderson, 2008]. However, while WCET estimations can be overly pessimistic and severely underuse processing capacity, scheduling SRT tasks by their mean execution times calls for overrun control methods to isolate tardiness effects that may otherwise hamper the schedulability of HRT tasks.

An implementation of RTWS for the Linux kernel, also proposed in Fonseca [2012], will be detailed further ahead in Section 4.3.8.

3.3.3.2 The p-CSWS Scheduler

To undertake the challenge of scheduling both HRT and SRT parallel real-time applications, Nogueira and Pinho [2012] recently proposed the Parallel Capacity Sharing by Work-Stealing (p-CSWS) scheduler, which complements M-CBS with a new capacity reclaiming mechanism able to exploit intra-task parallelism through WS. To the extent of our knowledge, p-CSWS is the first real-time scheduler to provide bandwidth isolation, residual capacity sharing, and support for intra-task parallelism, in a powerful and sophisticated algorithm for multi-core ORT systems.

p-CSWS extends the fork/join parallel real-time task model contemplated by RTWS to aperiodic and SRT tasks. Each task τ_i is assigned to a p-CSWS Dedicated Server (DS) S_i characterized by a budget C_i and a period T_i. For HRT tasks, C_i is set to the WCET, while for SRT tasks, the mean expected values for C_i and T_i are used. The bandwidth assigned to S_i is given by $U_i = \frac{Q_i}{T_i}$, and denotes the fraction of the capacity of one processor reserved by the server.

A set of n p-CSWS servers with total utilization $U_{sum} = \sum_i^n U_i$, can be scheduled through G-EDF on an identical multiprocessor platform comprised of m unit-capacity processors, if the conditions in 3.14 are met[22].

$$\begin{aligned} U_{max} &\leq 1 \\ U_{sum} &\leq m - (m-1)U_{max} \end{aligned} \tag{3.14}$$

At each instant, the following values are associated with a DS S_i: (i) its currently assigned deadline $d_{i,k}$, and (ii) its remaining execution capacity $0 \leq c_{i,k} \leq Q_i$. Each time a new job of τ_i arrives, it is enqueued in a First-In First-Out (FIFO) *job queue* of S_i. As in CBS, a server is said

[22] $U_{max} = max_{1 \leq i \leq n}(U_i)$ denotes the maximum server bandwidth.

to be *active* if it has pending work in its job queue, and *idle* otherwise. At any time instant t, the m active servers with earliest deadline are scheduled for execution in the m available processors, and referred to as *busy* servers.

When a job $j_{i,j}$ served by S_i is released at time t, causing the server to transition from idle to active, the test in Equation 3.15 is evaluated.

$$c_{i,k} < (d_{i,k} - t)U_i \tag{3.15}$$

If 3.15 holds, the newly arriving job is served with the current budget $c_{i,k}$ and deadline $d_{i,k}$ of S_i. Otherwise, a new instance $k+1$ of S_i is created, as $c_{i,k+1}$ is recharged to Q_i, and $d_{i,k+1}$ is set as $d_{i,k+1} = t + T_i$.

A busy server selects the job at the top of its queue for execution with a deadline equal to $d_{i,k}$. The budget $c_{i,k}$ of a busy DS S_i is decremented at each instant by the same amount. At time t, the reserved capacity in a p-CSWS server S_i is said to be *exhausted* if $c_{i,k} = 0$, and *expired* if $d_{i,k} < t$. When the capacity of a DS S_i is exhausted or expired, and there is pending work in S_i, a new server instance is established with $c_{i,k+1} = Q_i$ and $d_{i,k+1} = d_{i,k} + T_i$.

Dynamically generated threads are maintained in a local WS deque of the server where the job is currently being executed. Each server operates at the bottom of its deque, pushing and popping threads in a synchronization-free manner. Threads are successively dequeued from the bottom and assigned to the processor, until the deque is empty. At time t, if a DS S_i finishes the execution of its current job without exhausting the reserved capacity ($c_{i,k} > 0$), has no pending work in its job queue, and $c_{i,k}$ is greater than a lower bound Q_{min}, a new p-CSWS Residual Capacity Server (RCS) S_j^r of capacity $c_j^r = min(c_{i,k}, d_{i,k} - t)$ and deadline $d_j^r = d_{i,k}$ is dynamically generated.

A RCS is a p-CSWS server that applies a priority-based WS policy whenever its local deque is empty. When a RCS S_j^r is enqueued in the global queue, it competes for processor time like any regular active server with deadline d_j^r, and its capacity c_j^r can be used to execute any eligible thread through WS. Whenever a RCS S_j^r is selected for execution, it steals the topmost thread in the deque of earliest deadline active DS S_s in the system, with deadline $d_{i,k} \geq d_j^r$. A stolen thread executing in S_j^r consumes the remaining residual capacity c_j^r instead of that in its DS. A RCS is allowed to execute until its capacity is exhausted $c_j^r = 0$, and only prior to its deadline d_j^r. In other words, once a residual capacity has been exhausted or expired, it is is not replenished.

When the residual capacity of a busy RCS S_j^r is exhausted or expired, S_j^r remains active, but unable to execute, until all stolen threads in its local deque are reclaimed back by their respective DSs. To reclaim back stolen work, each DS maintains a list of stolen threads. If a busy DS finds its local deque empty, it looks at this list for stolen dedicated work, with the intent of reclaiming it back and selecting it for execution. Once an exhausted or expired RCS becomes idle[23], it is removed from the system.

p-CSWS Rules: The p-CSWS algorithm is compactly described with a set of rules defined in Nogueira and Pinho [2012], which we reproduce for quick reference:

[23] An idle server is one which has no pending work in its local deque. In this case, a RCS becomes idle once all stolen threads have completed, or been reclaimed back by their DSs.

Rule A: Whenever a server S_i changes its state from idle to active at some time t, a test is executed. If $c_{i,k} < (d_{i,k} - t)U_i$, no update of deadline and budget is necessary. Otherwise, $c_{i,k+1}$ is recharged to Q_i and the a new deadline is assigned $d_{i,k+1} = t + T_i$.

Rule B: Whenever a server S_i is selected for execution, it picks the bottom-most thread in its deque. While executing it, the budget $c_{i,k}$ is decreased by the same amount. If the server's capacity is either expired or exhausted, it is recharged to Q_i and its deadline $d_{i,k+1}$ is incremented by T_i.

Rule C: Whenever a server S_i finds its local deque empty, it verifies its thief list. If not empty, S_i follows the first pointer in the list, iteratively removing and executing those parallel threads until the list becomes empty.

Rule D: Whenever a server S_i completes its k^{th} job at time $t < d_{i,k}$, after having consumed $e_{i,j} < Q_i$ time units, and it has no pending work, a new residual capacity with capacity $min(c_{i,k}, d_{i,k} - t)$ and deadline $d_{i,k}$ is generated. S_i becomes idle and its remaining reserved capacity $c_{i,k}$ is set to 0.

Rule E: A new residual capacity less than a lower bound Q_{min} is assigned to the processor in which it was generated. The next active server with a later deadline, that executes on that processor, consumes the earliest deadline residual capacity prior to consuming its own dedicated capacity. When consuming a residual capacity, the server runs with the deadline of the residual capacity. If the processor idles beforehand, or if the capacity expires or is exhausted, it is disposed of.

Rule F: A new residual capacity consisting of at least Q_{min} is released to the global ready queue as a new RCS S_j^r with an execution capacity of $min(c_{i,k}, d_{i,k} - t)$ and deadline $d_{i,k}$. Whenever a RCS S_j^r is enqueued, it immediately competes for processing time as if it were a regular server with deadline d_j^r.

Rule G: If a RCS is selected for execution, it may only execute until time d_j^r, and the processor time c_j^r it receives is used to steal and execute the earliest deadline eligible thread with a current deadline at least d_j^r. Whenever a steal occurs, a pointer to the stolen task is added to the thief list of the stolen server.

Rule H: Whenever a thread is executed by a RCS S_j^r, it is scheduled using the residual capacity c_j^r and deadline d_j^r. As such, the execution capacity $c_{i,k}$ of its DS S_i remains unchanged. If the execution capacity of the RCS is either expired or exhausted, it is not recharged. If there is pending work, the RCS remains active. Otherwise, it is removed from the system.

Rule I: If a processor ever idles and there is any RCS in the global queue, then it dequeues the earliest deadline RCS and executes it without donating the resulting execution to any thread. The processor continues to execute the residual capacity server, as long as it would otherwise be idle, or the capacity of the RCS is neither exhausted nor expired.

p-CSWS has been tested with promising simulation results that show an effective decrease in the average tardiness of SRT tasks, when compared to M-CBS and M-CASH. Such results are primarily an effect of the powerful capacity-sharing and work-steling mechanism, able to boost execution times through an efficient exploitation of intra-task parallelism.

The clever design choice of achieving parallel execution, through WS, only when residual capacities are available, benefits system efficiency in two ways. WS controls the amount of active parallelism to avoid overly large scheduling, execution, and space overhead. By employing WS over residual capacities, it is expected that larger amount of concurrent servers generates more parallelism. In a sense, the bigger the system load, the more parallelism is exploited to help tasks respond faster.

In our opinion, p-CSWS is the first scheduling algorithm to meet the real-time requirements of a parallel ORT system, due to its ability to schedule both HRT and SRT computations, and efficiently harness processing power through parallel processing. To prove its practical feasibility, we decided to propose an implementation of the p-CSWS scheduler for the Linux kernel as detailed in Chapter , and analyze its performance through extensive tests described in Chapter .

3.4 Summary

Chapter 4

Linux Scheduling and Real-Time Support

"Making Linux GPL'd was definitely the best thing I ever did."

— Linus Torvalds

Linux, or GNU/Linux for the sake of neutrality[1], is a UNIX-like General-Purpose Operating System (GPOS) developed by Linus Torvalds, in 1991, for the Intel 80386 microprocessor. It quickly proved popular among eager kernel developers who, over the years, took the effort of improving what would become one of the most prominent Operating Systems (OSs) of today.

Released under the GNU General Public License (GPL), its source code is made fully available for anyone to use, study, modify and distribute. Today, the Linux kernel is maintained by an extensive team of programmers around the world, whose developments are still overseen by Linus Torvalds. Through years of cooperative progress, Linux has become the most compatible OS in the market[2], and steadily made its way into the most diverse areas of modern technology, from tiny embedded systems (such as smartphones and wristwatches), to televisions, personal computers, network servers, or massively-parallel supercomputers, to name a few.

Full access to such a complete, stable, flexible, and efficient OS, has also provided the academic and research communities with a powerful foundation for a large variety of projects. Although Linux is not a Real-Time Operating System (RTOS), it continues to serve the real-time community, not only as a testing platform, but also as the backbone for definitive implementations.

We begin this chapter with an architectural overview of the Linux kernel, and then take a technical look on Linux process scheduling internals. A presentation of the most notable projects tackling real-time support in the Linux kernel follows. Finally, we contextualize our work with respect to the opportunities for improvement in the area.

[1] The denomination has long been a source of controversy among members of the free and open-source software community.

[2] Supporting over 20 different processor architectures.

Unless otherwise stated, the content of this chapter refers to Linux v3.8.13, the version used for our implementation.

4.1 A Birdseye View of the Linux Kernel

An OS *kernel* is, in essence, a computer program which offers the basic abstraction between the hardware and user applications. Kernels usually abide by one of two major design patterns. *Monolithic kernels* are those implemented as a single process running in a single address space, and thus offering great simplicity, performance, and in-kernel communication. *Microkernels*, in turn, implement only the most elementary functions. Other features are delegated to several independent processes called *servers*[3], which run in distinct address spaces, and perform private functions which cannot be remotely invoked. Instead, servers communicate by exchanging[4] *service requests* via an internal Inter-Process Communication (IPC) message-passing mechanism [Love, 2010, Mauerer, 2008] (Section 4.1.2).

Linux falls closer to a monolithic kernel, but also adopts some microkernel characteristics. Contrarily to a pure monolithic kernel, it is divided into several schedulable and preemptible threads. Also, thanks to dynamically loadable *kernel modules*, secondary features, such as device drivers, can be activated or deactivated *on-the-fly*, and need not be included into the compiled kernel image [Love, 2010, Mauerer, 2008].

Like most modern OSs, the Linux kernel runs in a privileged level of execution called *kernel-space* (or *kernel-mode*), which involves a protected memory space and full access to hardware. User-level applications reside in *user-space* (or *user-mode*), where they cannot access kernel data structures or programs, nor directly interact with hardware devices.

4.1.1 System Calls and the User API

The kernel itself is not a process, but rather a process manager offering a set of well-defined kernel services that can be requested by way of *system calls*. User applications interact with the kernel via the System Call Interface (SCI) which, together with the *GNU C Library* (*glibc*), forms the user-space Application Programming Interface (API) used by developers to build Linux applications. Although system calls can be issued directly by the programmer, most interactions with the SCI are unwittingly carried out by higher-level *glibc* constructs. For example, after formatting and buffering the output data, `printf()` invokes the `write()` system call to request data be written to the console. When a system call is issued, the calling process momentarily switches to kernel-space, as the kernel enters *process context* to execute the service on its behalf. Upon completion, the kernel leaves process context, and the calling process returns to user-space.

[3] Not to be confused with the Constant Bandwidth Server (CBS) abstraction in real-time scheduling (Section 3.3.1.2)
[4] Between themselves and the microkernel.

4.1.2 Kernel Subsystems

The Linux kernel layer can be conceptually divided into 5 interdependent components, as depicted by Figure 4.1.

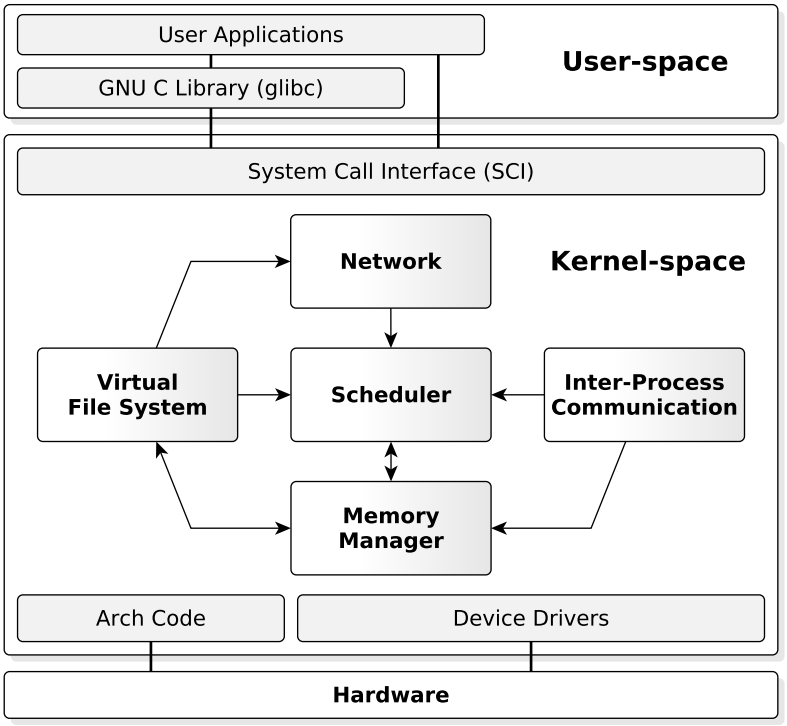

Figure 4.1: Overview of a Linux system

4.1.2.1 Process Scheduler

The *Process Scheduler* manages processes and their use of the available Central Processing Units (CPUs). The majority of its source code is kept in the `kernel/sched` directory of the kernel source tree. A more detailed overview of the process scheduler is presented in Section 4.2.

4.1.2.2 Memory Manager

The *Memory Management Subsystem* abstracts platform-specific details into one common memory interface, the *virtual memory*, which establishes the bridge between applications and the hardware Memory Management Unit (MMU). Among its main purposes and advantages, we point out the following:

- *Fair Memory Allocation* - Each process is given a fair share of the available physical memory.
- *Large Address Space* - Virtual memory can be much larger than the available physical memory, making it possible to run processes with very large memory requirements.

- *Memory Isolation and Protection* - Each process has its own virtual space, which cannot be overwritten by other applications.
- *Shared Virtual Memory* - Although memory is isolated across processes, they are allowed to explicitly share some portion of their memory.
- *Portability* - Programmers need not deal with physical memory organization, and can easily write machine-independent code.

The bulk of the Linux memory manager can be found in the `mm` directory of the source tree. [Bovet and Cesati, 2005, Kerrisk, 2010]

4.1.2.3 Virtual File System

The *Virtual File System (VFS)* provides a common abstraction for filesystems. It has the complex burden of defining a uniform interface to manipulate files, directories, and other objects, in the plethora of filesystems supported by Linux[5]. At the top layer, the VFS formalizes a set of common filesystem functions[6] such as `open()`, `close()`, `read()`, or `write()`. At a lower level, a group of plugins, in the `fs` directory of the source tree implement the aforementioned functions for each filesystem. [Kerrisk, 2010, Love, 2010, Mauerer, 2008]

4.1.2.4 Network Subsystem

The *Network Subsystem* abstracts both the devices and protocols used by Linux for inter-process communication over a network. Networking functionalities are exported to the user and other subsystems by way of the *socket interface*, invoked through the SCI. Sockets offer a standardized way to manage connections and transmit data across nodes, regardless of the underlying network and transport protocols. Lower level interaction between the subsystem and hardware device is made through the unified interface provided by the associated device driver. Networking sources are located in the `net` directory of the source tree. [Kerrisk, 2010, Mauerer, 2008]

4.1.2.5 Inter-Process Communication Subsystem

The *IPC Subsystem* provides the mechanisms by which concurrent Linux processes interact with each other[7]. The list below presents the most noteworthy.

- *Shared Memory* - The ability for processes to communicate by sharing a common physical memory space.
- *Locking Mechanisms* - Those used for concurrency control, such as *spinlocks*, *semaphores*, *atomic operations*, or *reader/writer locks*.
- *Message Queues* - A connection-less way to exchange messages of a specific data type.

[5] Linux supports over 40 filesystems, some of which can be rather unique and heterogeneous [Mauerer, 2008].
[6] By way of system calls in the SCI.
[7] Namely share resources, share or exchange data, and synchronize.

- *Signals* - A set of well-defined messages that can be asynchronously sent between processes.

Most of the IPC subsystem is implemented in the `ipc` directory of the kernel source tree. For more information on this topic, please refer to the annexed bibliography. [Bovet and Cesati, 2005, Kerrisk, 2010, Mauerer, 2008]

4.1.3 Architecture Specifics and Device Drivers

Linux kernel developers lay a great deal of emphasis on portability. While most kernel code is architecture-independent, a number of low-level elements[8] trade portability for optimal code specifically tuned for each architecture. The `arch` directory of the source tree contains most of the architecture-dependent code in Linux, categorized into 29 sub-directories[9]. [Love, 2010, Mauerer, 2008]

With the exception of the CPU, memory, and a few other elements, all device control operations are performed via a *device driver*. Linux device drivers are commonly implemented as individual kernel modules, containing specific code to coordinate the affected physical device. The goal is to provide a unified interface between the kernel and a certain type of device. Overall, the device driver manages both sides of the data transfer mechanism of a certain device. It (1) effectively operates the associated device, in accordance to high-level commands originated in the kernel, and (2) handles and translates raw data sent by the device to the OS. [Bovet and Cesati, 2005, Love, 2010, Mauerer, 2008, Venkateswaran, 2008]

Linux classifies devices (and device drivers) into 3 distinct types: [Love, 2010, Venkateswaran, 2008]

- *Character Devices*, *char devices*, or *cdev*, are those accessed sequentially by the OS, *i.e.* data is typically transferred as a stream of characters (byte-by-byte). A wide range of devices fall into this category, from keyboards to printers, sound cards, or any device connected to a serial or parallel port.

- *Block Devices*, or *blkdev*, are addressable in device-specified blocks, and support random access of data. These are media storage devices capable of hosting a filesystem, such as hard drives, floppy drives, or USB memory sticks.

- *Network Devices*, or *Ethernet Devices*, refer to physical adapters that connect the system to a network. Network device drivers differ from char and block drivers in that they are accessed via the socket interface, rather than represented and operated as regular files at the filesystem level.

Comprehensibly, device drivers play a crucial role in some of the subsystems presented in Section 4.1.2, particularly in the VFS and network subsystems, where they provide the basic hardware abstraction to implement the unified interfaces exported to user-space.

[8] The process scheduler, for example, uses architecture-dependent code to perform context switching.
[9] One for each platform supported by Linux

The source code for the set of device drivers distributed with the Linux kernel can be found in the `drivers` directory of the source tree.

4.2 The Process Scheduler

In Linux, the process scheduler is the subsystem responsible for coordinating the allocation of processor time. The scheduler implements a set of rules, referred to as *scheduling policy*, to decide upon when, and how, processes are selected for execution. The Linux scheduler foresees the existence of several active processes simultaneously contending for processor time. As a *multitasking* OSs, it gives the illusion of simultaneous processing through a dynamic *time-sharing* scheduling policy that switches between executions at very short intervals, in an attempt maximize resource utilization and share CPU time as evenly as possible among running processes.

The current Linux scheduler was designed by Ingo Molnar, and introduced in Linux v2.6.23 to replace the original O(1) scheduler. In general, it consists of an extensible collection of scheduler modules called *scheduling classes*, coordinated by the *core scheduler*.

Linux makes no major distinction between processes and threads[10], in the sense that they are both internally represented by the same data structure. Henceforth, we refer to both processes and threads as *tasks*.

4.2.1 Linux Tasks

Each task in the Linux kernel is represented by an instance of the extensive `task_struct` data structure declared in `include/linux/sched.h`, also known as the *process descriptor*. A `task_struct` object maintains the necessary data to manage and perform scheduling decisions upon a task. All tasks, or more precisely their process descriptors, are kept in a system-wide circular doubly linked list called the *task list*. Among the `task_struct` attributes relevant to the scheduler, we will pay special attention to those listed and described in Table 4.1.

Attribute	Description
`pid`	*Process ID* specific to each task
`tgid`	*Group ID* shared by all child threads created by the same task
`state`	The current state of a task
`flags`	Additional information about special characteristics of the task at a given moment
`prio/normal_prio`	Dynamically computed priorities
`static_priority`	Relative priority assigned at configuration time
`rt_priority`	Static priority for a SCHED_RR or SCHED_FIFO task (Section 4.2.4)
`sched_class`	The currently associated scheduling class
`policy`	The currently associated scheduling policy
`on_cpu`	Binary attribute to indicate whether the task is executing or not
`on_rq`	Binary attribute to indicate whether the task is on the list of ready tasks or not
`nr_cpus_allowed`	Number of CPUs that can execute the task.
`cpus_allowed`	Bit mask representing the CPUs where the task can execute

Table 4.1: A list of relevant `task_struct` attributes.

Each task is bound to exactly one scheduling class and policy at any given time, but allowed to transition to another class or policy via the `sched_setscheduler()` system call, passing

[10] A process is an instance of a computer program, while a thread is a smaller unit of control that is part of a process.

the new policy identifier as a parameter. Scheduling classes and policies are presented in Section 4.2.4.

Linux scheduling is not restricted to individual tasks, but rather *scheduling entities*. A scheduling entity represents either a single task or a group of tasks recognized by the scheduler as a single entity. Scheduling classes define their own scheduling entity data structures, with the needed attributes to perform class-specific scheduling operations. These structures typically encompass statistical elements, group scheduling attributes, or current and historical task details[11]. To schedule tasks individually, the process descriptor contains an instance of the scheduling entity data structure defined by each scheduling class. We will refer back to this aspect in Section 4.2.4.

Restricting our study to task scheduling, we address task and entity scheduling interchangeably.

4.2.1.1 Task States and Transitions

To describe what is currently happening to each task, a total of 9 task states for the `state` attribute of the process descriptor are defined in `include/linux/sched.h`. We emphasize 5 major states depicted by Figure 4.2.

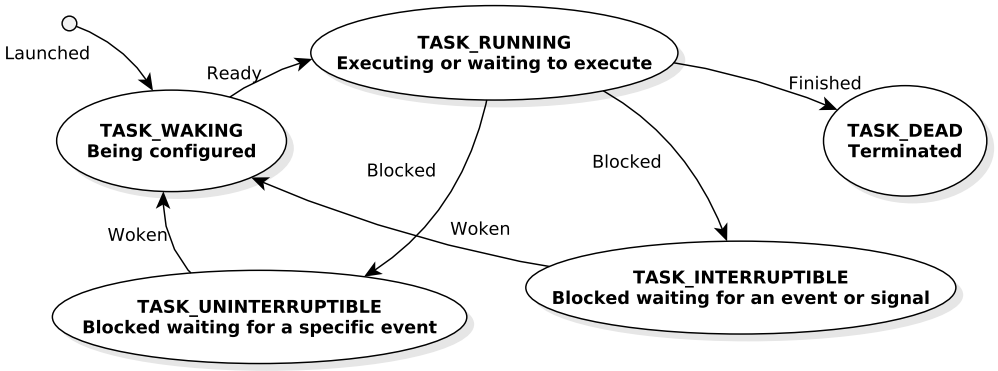

Figure 4.2: Linux task states and transitions

Throughout the startup process, a task exhibits the `TASK_WAKING` state, only transitioning to `TASK_RUNNING` when it becomes *runnable*, *i.e.* fully configured and *ready* to execute. Any runnable task, either executing or waiting for execution, remains in the `TASK_RUNNING` state until it blocks[12]. A long as a task is *blocked*, holding the `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state, it cannot be scheduled for execution. When a task is set to wake up from a blocked state, it switches to `TASK_WAKING`. Upon completion of the wake up process, it returns to `TASK_RUNNING` and is ready to execute. Once a task finishes execution, its state is set to `TASK_DEAD` before being removed from the system.

Section 4.2.5.3 explains how state transitions are handled and effectuated by the Linux scheduler.

[11] Such as the total elapsed execution time, a time stamp for the start of execution, and other data specific to the scheduling class

[12] Due to an I/O request, voluntarily via call to a specific kernel function, to wait for an interrupt, *et cetera*.

4.2.1.2 Task Affinity

In multiprocessor systems, `cpus_allowed` is used to set the affinity of a task. The system call `sched_setaffinity()` allows the user to specify a subset of CPUs eligible to execute a task. To perform a particular operation, the kernel may also temporarily bind a task to a specific CPU. It does so by forcefully setting a single bit on the CPU mask, which is later restored back to the previous value once the operation finishes. At any given time, `nr_cpus_allowed` reflects the number of eligible CPUs set in `cpus_allowed`.

4.2.2 Runqueues

Runnable tasks are kept in CPU-specific queues, called *runqueues*. The runqueue is a primordial element of the scheduler, defined in `kernel/sched/sched.h` by the `rq` data structure.

The Linux scheduler abides by a semi-partitioned architecture. Each CPU maintains an independent runqueue of local ready tasks, such that a task is not allowed to exist in more that one runqueue or CPU at a given time. Load balancing is achieved through an exchange of tasks between runqueues, which immediately implies migration across processors.

The pivotal attributes of the `rq` structure are listed next:

- `lock` defines a spinlock to control concurrent access to the runqueue.
- `nr_running` maintains a counter for the number of ready tasks in the runqueue.
- `cpu` stores the ID of the CPU assigned to the runqueue.
- `curr` points to the process descriptor of the task that is currently executing in the local CPU.
- `clock` keeps track of the local CPU time.
- `sd` points to the associated scheduling domain (Section 4.2.6.1).

To ensure data consistency, runqueues are protected by a *spinlock*. All runqueue manipulations must acquire the spinlock `lock` beforehand. In special situations, when multiple runqueue locks are required, one must follow the rule established by the development team to avoid potential deadlocks. It states that runqueue locks are to be acquired in order of increasing memory address of the `rq` object. `double_lock_balance()` and `double_unlock_balance()` provide a means to lock and unlock a second runqueue[13] straightforwardly. However, to respect the locking rule `double_lock_balance()` may drop the held lock[14], requiring special caution on the part of the programmer.

Following the modular design of the scheduler, tasks are not directly stored in the `rq` data structure, but in specific data structures defined for each scheduling class and instantiated in `rq`. These data structures will be addressed in Section 4.2.4.

[13] We refer to the scenario where one wants to acquire a runqueue lock while holding another.

[14] If the second runqueue has a lower memory address, the held lock must be dropped and acquired again after the second runqueue is locked.

4.2.3 Generic Data Types Relevant to Scheduling

For the sake of code simplicity and reuse, the Linux kernel source provides its own APIs to maintain structured data elements.

Scheduling classes[15] use either *doubly linked lists* or *red-black trees* to store and organize runnable tasks, according to their priorities, in class-specific runqueues. Every data structure has operation costs, and the structural design of a runqueue has a tremendous impact on scheduling efficiency. The data structure of choice must be in tune with the intervening algorithm (the scheduling policy), so as to enable quick scheduling decisions at a low maintenance effort.

4.2.3.1 Doubly Linked Lists

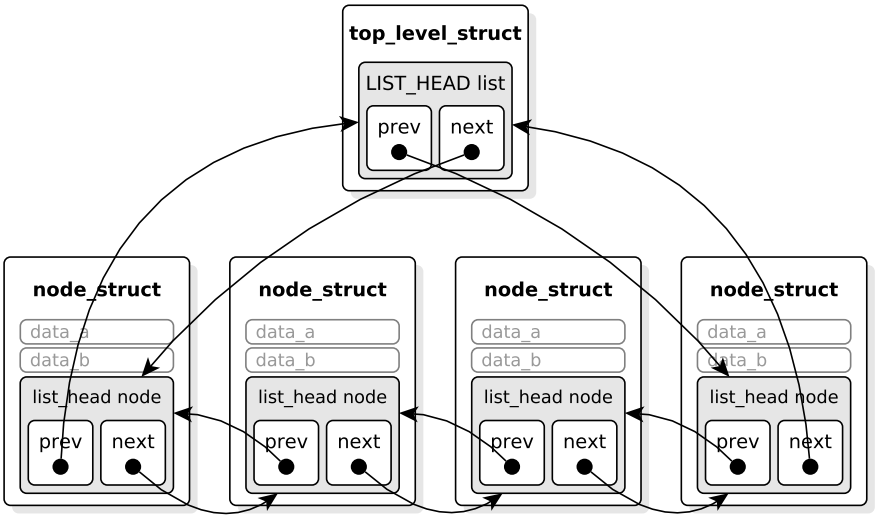

Figure 4.3: Doubly linked list example

Traditional linked lists are defined through a custom-built data structure containing the data to be linked as well as pointers to adjacent nodes of the same data structure. Linux offers a more general interface, in which nodes are set up by a `list_head` object embedded into another data structure, as shown by Figure 4.3. The `list_head` data structure (Listing 4.1) declared in `include/linux/types.h` simply defines two pointers to the previous and next element of the list (also of type `list_head`).

```
1 struct list_head {
    struct list_head *next, *prev;
3 };
```

Listing 4.1: The list_head data structure.

`include/linux/list.h` provides a set of routines to manage linked lists, namely to initialize, add, remove, and traverse chains of `list_head` elements. Seeing that `list_head`

[15] The default scheduling classes in Linux v3.8.13.

nodes are nested inside the structure that contains the data, accessing the enveloping data structure requires an additional step that can be carried out with the `list_entry(ptr, type, member)` macro, where `ptr` is a pointer to the `list_head` element, `type` is the type of the enveloping data structure, and `member` is the name of the `list_head` variable within the data structure.

```
1 #define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

Listing 4.2: The list_entry macro.

As seen in Listing 4.2, `list_entry` simply abstracts the `container_of` macro, which retrieves the address of the enclosing `type` object containing `ptr`, based on the offset of `member`.

On a list of n unsorted nodes, insertions and deletions are performed in constant $O(1)$ time, but search operations are subject to $O(n)$ complexity. On sorted linked lists, insertions are also performed in linear $O(n)$ time. [Love, 2010, Mauerer, 2008]

4.2.3.2 Red-Black Trees

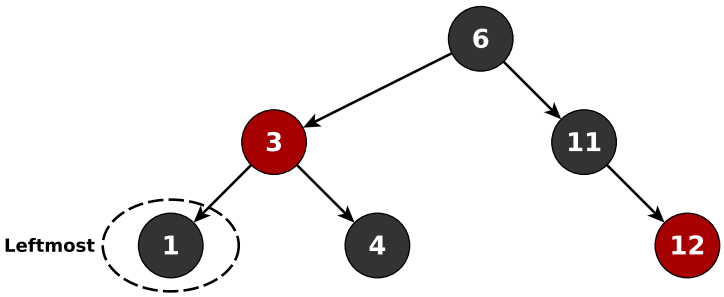

Figure 4.4: Red-black tree example

A *red-black tree* is a type of self-balanced binary search tree, whose nodes are sorted by non-decreasing order of a predefined *key*. The peculiarities and complex management of red-black trees are abstracted by the generic API coded in `include/linux/rbtree.h`.

A red-black tree can be declared with an instance of the `rb_root` object provided by the API. Similarly to linked lists, a `rb_node` object can be declared within a data structure to specify a red-black tree node. `rb_entry(ptr, type, member)` (Listing 4.3) is the `list_entry` equivalent for red-black trees, to access the containing data structure of a red-black tree node.

```
#define rb_entry(ptr, type, member) \
2   container_of(ptr, type, member)
```

Listing 4.3: The rb_entry macro.

The leftmost node of a red-black tree is always the one with the lowest key value, and accessible in constant $O(1)$ time when cached[16]. Though complex, red-black trees offer good worst-case

[16]When a reference to the leftmost node is maintained

performance. In a tree of n nodes, manipulations[17] are guaranteed in logarithmic $O(log(n))$ time. [Love, 2010, Mauerer, 2008]

4.2.4 Scheduling Classes and Policies

In practice, each scheduling class is an instance of the generic `sched_class` data structure defined in `include/linux/sched.h`. The `sched_class` structure provides a set of hooks, or function pointers, for well-defined scheduling operations invoked by the core scheduler. All 4 scheduling classes in the kernel are organized hierarchically by way of a pointer (the `next` attribute of `sched_class`) referring to the next class in the hierarchy.

Creating a new scheduling class, is therefore a matter of instantiating a new `sched_class` structure, integrating it in the hierarchy, and defining the set of functions required to perform scheduling decisions. Next, we present a list of the most pertinent hooks of `sched_class`, along with a short description of their purpose and semantics:

- `enqueue_task` inserts a task onto the runqueue.
- `dequeue_task` removes a task from the runqueue.
- `yield_task` is invoked when the current task voluntarily yields the CPU.
- `check_preempt_curr` checks if the current task should be preempted by another task.
- `pick_next_task` selects and returns the next task[18] to be executed.
- `put_prev_task` is invoked before a context switch, to manage the task leaving the CPU.
- `set_curr_task` is mostly invoked when the policy or priority of a task changes.
- `task_tick` is invoked periodically by the core scheduler, according to the frequency defined by the `HZ` macro.
- `task_fork` is invoked when an executing task spawns another task (thread).
- `switched_from` and `switched_to` are invoked with different parameters, when a task changes from a scheduling class to another.
- `prio_changed` is invoked when the priority of a task changes.
- `select_task_rq` selects and returns the CPU where the task shall be scheduled for execution. Invoked whenever a task wakes up.
- `pre_schedule` is invoked by the core scheduler before any scheduling decision.
- `pre_schedule` is invoked by the core scheduler after a scheduling decision.
- `task_waking` is invoked when a task is set to wake up.

[17] Insertions, arbitrary searches, and deletions.
[18] The highest priority task in the class.

- `task_woken` is invoked at the end of the wake up routine.
- `set_cpus_allowed` is invoked when the affinity[19] of a task changes.

The last 6 functions in the list, from `select_task_rq` to `set_cpus_allowed`, are specific of multiprocessor configurations and only defined when the kernel is compiled with Symmetric Multiprocessing (SMP) support.

Each scheduling class encloses specific logic to implement one or more scheduling policies. Each task in the system is bound to exactly one class and policy via the `sched_class` pointer and `policy` attribute of the process descriptor (Section 4.2.1).

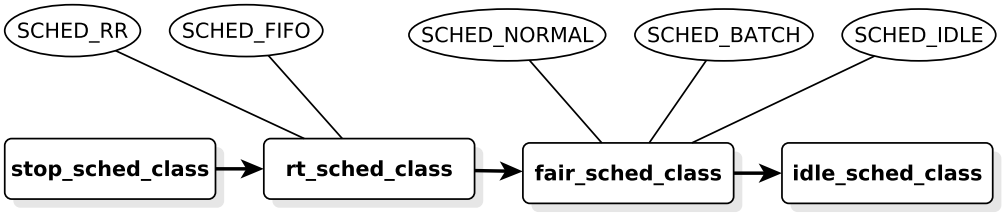

Figure 4.5: Hierarchy of scheduling classes and policies

Linux[20] specifies a total of 5 scheduling policies identified by a set of macros defined in `include/uapi/linux/sched.h` and reproduced in Listing 4.4. These policies are defined by two classes of the scheduling class hierarchy depicted in Figure 4.5.

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
```

Listing 4.4: Scheduling policy identifiers.

4.2.4.1 The Stop Class (stop_sched_class)

The `stop_sched_class` is a special scheduling class, outlined in `kernel/sched/stop_task.c` to manage the per-CPU *stop task*. The stop task is the highest priority task in the kernel, as it can preempt anything and not be preempted. It is used to force critical operations on a given CPU, usually related to load balancing and CPU *hotplugging*. For example, when an executing task changes its CPU affinity, if the current CPU (associated to the task) is disallowed in its `cpus_allowed` bit mask, the stop task ensures it is migrated to another CPU.

[19] The set of CPUs eligible to execute a task.
[20] As of Linux v3.8.13.

4.2.4.2 The RT Class (rt_sched_class)

The rt_sched_class is defined in kernel/sched/rt.c to implement two soft real-time policies:

- **SCHED_FIFO** entities are scheduled in a First-In First-Out (FIFO) manner, and each entity is allowed to execute as long as it voluntarily decides to relinquish the processor.
- **SCHED_RR** implements a preemptible Round-Robin (RR) policy, where each entity is assigned a *time slice* and allowed to execute until its time slice is exhausted. Once an entity exhausts its time slice, it is preempted by the highest priority entity awaiting execution. SCHED_RR entities are also preempted by any SCHED_FIFO entity in the system.

According to the descriptions in Sections 4.2.1 and 4.2.2, the RT class extends rq and task_struct with its own runqueue and scheduling entity designs, respectively.

Each RT runqueue is an instance of the rt_rq structure in coded in kernel/sched/sched.h. The list of ready entities is specified by an object of type rt_prio_array within the rt_rq structure. rt_prio_array sets up a queue where each node is a list of entities with the same priority, along with a related bitmap for optimized access to the queue.

The rt_sched_entity, stated in include/linux/sched.h, extends task_struct with support for RT scheduling entities. It defines the essential attributes for each SCHED_FIFO and SCHED_RR entity, such as the time slice, a list_head element to place it in the RT runqueue, or a pointer to the currently associated runqueue.

Note that the concept of *soft real-time* presented here differs from definition considered in this thesis and the real-time task model formalized in Chapter 3 (Section 3.1). rt_sched_class does not foresee the existence of specific timing constraints on the release and response times of scheduling entities. Instead, each entity is assigned a numeric priority (ranging from 0 to 99), which defines its urgency over other executions, but does not offer the response-time determinism of a deadline-based model. Although the schedule provided by rt_sched_class prioritizes SCHED_FIFO and SCHED_RR entities over other tasks in the system, it cannot provide any precise timing guarantees nor schedule tasks by their specific timing parameters, as it would be expected from a real-time scheduler.

4.2.4.3 The Completely Fair Scheduler (fair_sched_class)

fair_sched_class implements the Completely Fair Scheduler (CFS) in kernel/sched/fair.c, by way of 3 scheduling policies:

- **SCHED_NORMAL** is the default scheduling policy for general-purpose interactive programs, following the idea of an equal distribution of processor time among ready scheduling entities.
- **SCHED_BATCH** is the policy defined by the CFS to schedule CPU-intensive batch processes in the background. SCHED_BATCH entities are scheduled much like SCHED_NORMAL, but

in order to prevent them from affecting the execution of interactive processes, they are not allowed to preempt SCHED_NORMAL entities.

- **SCHED_IDLE** schedules entities of minimal importance, only allowed to run when a CPU would otherwise be idle.

fair_sched_class scheduling entities are defined by the sched_entity data structure in include/linux/sched.h. SCHED_NORMAL entities are scheduled by their *virtual runtime*. The virtual runtime of each entity is supported by the vruntime attribute of sched_entity, and computed as function of its elapsed execution time and priority. CFS priorities range from 100 (highest) to 139 (lowest), and do not define the order by which entities are scheduled, but rather the preference of an entity over the remainder. The higher the priority of an entity, the slower its virtual runtime grows. Therefore, since CFS entities are scheduled by vruntime, higher priority entities receive larger shares of processing time.

Another crucial element of the CFS is the cfs_rq data structure, declared in kernel/sched/sched.h to extend the rq structure with special CFS attributes. tasks_timeline defines the per-cpu red-black tree of ready CFS entities, ordered by non-decreasing vruntime. The majority of scheduling decisions are based upon the currently executing entity and the one with the lowest vruntime. To access the entity with the lowest vruntime in constant time, rb_leftmost keeps a reference to the leftmost node of the red-black tree[21].

The virtual runtime of an executing entity is updated at every scheduler tick. Towards a fair distribution of work, as soon as the executing entity no longer has the lowest vruntime value it relinquishes the processor to an entity of higher priority. In practice, due to scheduling overhead and granularity constraints, no schedule obtained by the CFS is perfectly fair. To reduce the total amount of scheduling overhead and provide a reasonably fair distribution of processor time, it defines a *scheduling latency* value and a *minum granularity* value, used in combination to enforce a lower execution bound[22].

4.2.4.4 The Idle Class (idle_sched_class)

The idle_sched_class is a special scheduling class, declared in kernel/sched/idle_task.c to manage the kernel *idle task* (also known as the *swapper task*). The per-CPU idle task is scheduled for execution when no other ready tasks exist locally. The idle task is the lowest priority task in the system, and its sole purpose is to keep a processor busy when it runs out of work.

Even though their designations may be misleading, SCHED_IDLE and idle_sched_class are not one and the same thing. As stated in Section 4.2.4.3, SCHED_IDLE is a CFS policy to schedule low priority executions.

[21] Since the tree is sorted in non-decreasing order of vruntime, the leftmost node holds the entity with lowest vruntime.

[22] Before which, entities cannot be preempted

4.2.5 The Core Scheduler

So far we have seen how scheduling policies are implemented within a scheduling class. However, since scheduling classes merely establish how scheduling operations are to be performed in the context of a particular policy, it falls upon the core scheduler to carry out the most elementary scheduling operations and effectively assign tasks to the CPU.

As stated in Section 4.2.4, the Linux scheduler formalizes a set of conventional operations, as a collection of function pointers in the `sched_class` data structure. Scheduling classes instantiate the `sched_class` structure and define the set of scheduling class hooks, by assigning each function pointer a routine that implements the respective operation. Under this general but well-defined structural model, the internals of each scheduling function vary between classes, but the semantics remains the same. While this structural design provides a common and transparent interface between the core scheduler and scheduling modules, it also allows the configuration of new scheduling classes without the need to completely reformulate the scheduler.

The core scheduler is outlined by a vast number of routines defined in `kernel/sched/core.c`. In the remainder of this section, we direct our attention to the most important.

4.2.5.1 The Periodic Scheduler - scheduler_tick()

The *periodic scheduler* is implemented by the `scheduler_tick()` function, which is called periodically by the internal kernel timer at a frequency stipulated by the `HZ` macro. In SMP configurations, `scheduler_tick()` is called independently on each active CPU.

`scheduler_tick()` updates the local runqueue clock, before invoking the `task_tick` hook of the scheduling class associated to the currently executing task. `fair_sched_class`, for example, takes this opportunity to verify if the current task should be preempted by another task. If so, it internally[23] sets the `TIF_NEED_RESCHED` flag in the process descriptor of the current task, to indicate that it should leave the processor. Of course, this example refers to the internal implementation of `task_tick` for a specific scheduling class (`fair_sched_class`), and is not to be taken as mandatory for all classes.

4.2.5.2 The Main Scheduler - __schedule()

The *main scheduler*, specified by the `__schedule()` function, can easily be described as the heart of the Linux scheduler. Like `scheduler_tick()` it is called locally (on each CPU), by way of its wrapper `schedule()`, to replace the currently executing task by another one, *i.e.* pick the highest priority local task and assign it to the CPU.

As described ahead, the modular design of the scheduler is put to good use in `__schedule()`, as it defers most of its work to the configured scheduling classes.

`schedule()` is invoked by kernel routines exhibiting the `__sched` prefix, and as a consequence of 3 special events [Seeker, 2013]: (i) when the currently executing task blocks, (ii)

[23] Inside the `task_tick` hook of `fair_sched_class`.

when a blocked task wakes up, or (iii) when the TIF_NEED_RESCHED flag is set on the currently executing task.

To ensure atomic execution, __schedule() starts by disabling preemptions. Then, it fetches the local CPU ID, runqueue, and sets the prev pointer to the currently executing task that is set to yield the CPU. To handle possible races, if prev is in the TASK_INTERRUPTIBLE state, and has a pending signal, its state is set back to TASK_RUNNING[24]. Otherwise, a call to deactivate_task() removes the task from the runqueue. activate_task() and deactivate_task() are the two major functions used by the core scheduler to add and remove tasks from the runqueue, using the enqueue_task and dequeue_task hooks of the associated scheduling class.

Before the next task is selected for execution, the pre_schedule and put_prev_task hooks in the scheduling class of prev are invoked. Scheduling classes may take this opportunity to keep statistics up to date, or perform specific operations[25] before the next task is picked.

The scheduler then selects the next task for execution. It queries each scheduling class, by their order in the hierarchy, invoking the pick_next_task hook until a process descriptor is returned. Therefore, tasks assigned to lower priority scheduling classes do not execute unless higher priority classes run out of work. The logic by which tasks are selected for execution is inherent to the internal implementation of the pick_next_task hook in each scheduling class. Once the pick_next_task method of a queried class returns, the next pointer (in __schedule()) is set to reference the returned process descriptor. If no task has been returned by the time the idle_sched_class is queried, then the swapper task is picked for execution[26].

At this point, and after clearing the TIF_NEED_RESCHED flag of prev, if prev and next point to distinct tasks, a *context switch* is performed. A call to context_switch() effectively replaces prev by next in the local CPU.

If the post_schedule variable of the local rq structure is set, __schedule() completes with a call to the post_schedule hook of the scheduling class associated with next. As described in Section 4.2.6.4, post_schedule is used by the rt_sched_class, in SMP configurations, as part of its load-balancing mechanism [Seeker, 2013].

4.2.5.3 Task Blocking, Waking, and Termination

In Linux, a blocked task waiting for a certain condition or event is kept in a *waitqueue* associated with that condition or event. A task blocks voluntarily by way of a specific system call that sets its state to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE, and invokes schedule() to deactivate it and take it off the CPU. Once a condition or event occurs, the scheduler wakes up every task of the associated waitqueue.

Task activation is performed by the *wake up* family of functions wake_up_new_task(), try_to_wake_up(), and try_to_wake_up_local(), of kernel/sched/core.c. First,

[24] __schedule() may be called in a deferred fashion. As such, a task that is set to block may receive a signal before it is actually taken off the CPU. Recall that only tasks in the TASK_INTERRUPTIBLE state can be woken up by a signal (Section 4.2.1).

[25] For example, rt_sched_class uses pre_schedule as a part of its load-balancing scheme, to pull a high priority task from another runqueue [Seeker, 2013].

[26] idle_sched_class is the last class in the hierarchy. It always returns the idle, or swapper, task.

they switch the task state to `TASK_WAKING` invoke the `task_waking` method of the associated scheduling class. A subsequent call to the `select_task_rq` hook returns the CPU ID where the task is to be placed. Then, a call to `ttwu_queue()` triggers `activate_task()` to insert the task onto the runqueue of the associated CPU, switch the task state to `TASK_RUNNING`, and call the `task_woken` method of the scheduling class. From this moment on, the task can be selected for execution by the the associated policy.

Whether a task dies involuntarily, or by direct invocation of the `exit()` system call, the termination event is handled by the `do_exit()` routine of `kernel/exit.c`. `do_exit()` sets the `PF_EXITING` flag on the `flags` attribute of the process descriptor and, after freeing the resources allotted to the task, invokes the main scheduler to switch the task off the CPU.

For detailed information on the topics addressed in this section, we refer the reader to Aas [2005], Bovet and Cesati [2005], Love [2010], Mauerer [2008], Seeker [2013]

4.2.6 SMP Support and Load-Balancing

As discussed in Chapter 2, the efficiency of a multiprocessor scheduler is directly tied with its ability to keep the workload balanced across CPUs. Dynamic scheduler achieve load-balancing through task migration *on-the-fly*, but migrations have inherent costs that may overpower the benefits of maintaining an even distribution of work.

Linux supports a wide range of heterogeneous processor architectures, from uniprocessor platforms, to hyperthreading cores[27], multi-core chips, or large and complex Non-Uniform Memory Access (NUMA) configurations. While hyperthreading units share the same memory and cache, each core in a SMP machine has its own cache space. On NUMA machines, memory modules are typically distributed across nodes, and memory access speeds vary depending on the proximity of the CPUs to the memory.

The effects of task migration on each of these architectures can be quite distinct and unpredictable. For instance, even on SMP systems cache misses may cause a greater decrease in performance than scheduling tasks on busy CPUs. On NUMA platforms, moving tasks across distant nodes is expectedly more expensive than across tightly-coupled units on a local node, and tasks should not be pushed off a NUMA node while local units[28] remain idle.

4.2.6.1 Scheduling Domains

To cope with the plethora of supported multiprocessor topologies efficiently, the kernel organizes adjacent CPUs into logical *scheduling domains*, structured hierarchically in consonance with the physical architecture. Scheduling domains may envelop other domains, as well as *scheduling groups* which either represent individual processing units or clusters of CPUs treated as one within the domain. [Ker, c, Aas, 2005, Bovet and Cesati, 2005, Mauerer, 2008, Seeker, 2013]

Small multi-core schemes, as considered in this work, typically entail a single scheduling domain with one scheduling group per core. Hyperthreaded cores originate an additional pair

[27] Each hyperthreading unit is treated by the Linux kernel as distinct CPU.
[28] Other CPUs on that node.

of scheduling groups, one for each logical hyperthreading unit. Each node of a NUMA configurations is represented by a scheduling domain containing all nested domains and groups. The top-level domain encloses the complete hierarchy of configured domains and groups.

Through this hierarchy of scheduling domains, the scheduler can be tuned to balance the load more frequently on higher level domains, across neighboring units, and thereby have a greater control over global overheads. A *balancing policy*, together with a set of *policy flags*, defined for each domain, specify a series of balancing criteria such as the frequency of load-balancing attempts, the imbalance threshold that triggers a load-balancing operation on the domain, a bound on how far up the domain hierarchy tasks of the domain are allowed to migrate, what to do when one of the associated CPUs becomes idle, *et cetera*. [Seeker, 2013]

Scheduling domains and groups are defined in `include/linux/sched.h` and automatically configured by the kernel at startup time. Scheduling domains are instances of the `sched_domain` descriptor. Scheduling groups are represented by the `sched_group` data structure instantiated in `sched_domain`. By way of a pointer to the next element, `sched_group` sets up a circular linked list to connect all groups of the domain. Each runqueue is bound to exactly one scheduling domain via the `sd` pointer of the `rq` structure.

4.2.6.2 Cpusets and Root Domains

In Section 4.2.1 we have mentioned that Linux can schedule groups of tasks internally represented as scheduling entities. Kernel Control Groups (CGroups) allow users to agglomerate tasks into hierarchical groups and assign them specific shares of the available resources, in what is known as *group scheduling*. In a nutshell, when a quota of the processing bandwidth is allocated to a CGroup, the scheduler distributes this share among all scheduling entities on the CGroup (nested tasks or CGroups) [Ker, a, Mauerer, 2008].

Besides processing time, users can define subsets of the available CPUs and memory, called *cpusets*, and appoint them for the execution of certain tasks or CGroups. Multi-level cpuset hierarchies can be established, and unrelated cpusets are said to overlap if they share common CPUs or memory nodes. Disjoint cpusets, which do not overlap with other cpusets in the system[29], can be marked as *exclusive cpusets* to monopolize the allotted CPUs and memory Ker [b].

As explained in Section 4.2.6.4 the `rt_sched_class` uses global information to perform load-balancing decisions. With an increasing number of CPUs, contention on these global resources is bound to degrade the performance of the system, imposing a scalability bottleneck. To limit the span of global RT decisions, once an exclusive cpuset is created an isolated domain, called *root domain*, is defined and attached to that cpuset[30]. A root domain is an instance of the `root_domain` data structure formalized in `kernel/sched/sched.h`. It contains a bitmap of overloaded RT runqueues[31], as well as a priority bit array to track the priority of each CPU[32] executing RT tasks. By default, a single root domain instance maintains system-wide data about

[29] Except with the ascendants or descendants on their hierarchy.
[30] The scope of any scheduling decision on an exclusive cpuset is restricted to the subset of associated CPU.
[31] Runqueues holding more than one RT entity.
[32] At each instant, the priority of a CPU is equal to that of its highest priority task.

all CPUs available in the system. [Seeker, 2013]

4.2.6.3 CFS Load-Balancing

In `fair_sched_class` a three-stage load-balancing mechanism is in effect.

The *active*, or *periodic*, mechanism is implemented in `run_rebalance_domains()`, triggered by `scheduler_tick()`[33]. `run_rebalance_domains()` goes up the scheduling domain hierarchy, starting with the domain of the current CPU, and checking if each domain should be balanced[34]. If so, a call to `load_balance()` finds the busiest scheduling group (if any) on the domain, then the busiest runqueue (if any) on the busiest group. After finding the busiest runqueue, `move_tasks()` pulls a predetermined number of tasks[35] to the local CPU. Once the desired volume of work is pulled, the mechanism completes. [Aas, 2005, Mauerer, 2008, Seeker, 2013]

The *idle balancing* stage is defined in `idle_balance()`, and invoked by `__schedule()` whenever the current CPU is about to become idle. The work performed by `idle_balance()` is similar to that of `run_rebalance_domains()`, as it iterates through the hierarchy of scheduling domains trying to pull tasks from overloaded runqueues. [Bovet and Cesati, 2005, Seeker, 2013]

The third stage is outlined in the `select_task_rq` method of `fair_sched_class`, specifically `select_task_rq_fair()`, which defines the placement of an awakening task. Here, unless the task is pinned to its previous CPU, the hierarchy of scheduling domains is traversed[36] in search of a domain that is set to be balanced[37]. A search for least loaded group on the domain follows, and the idlest (an idle or the least loaded) CPU of that group is returned. [Seeker, 2013]

4.2.6.4 RT load-balancing

Rather than interactivity and resource utilization, `rt_sched_class` strives to satisfy the order of execution by keeping the highest priority tasks executing on the configured CPUs as much as possible. To achieve this, each CPU proactively reacts to load and priority changes through push and pull operations, which continuously redistribute waiting tasks to other CPUs where they can execute straight away. [Seeker, 2013]

Push operations are attempted by the `push_rt_tasks()` routine of `kernel/sched/rt.c`, which calls `push_rt_task()` to push the highest priority waiting task off the local runqueue [Seeker, 2013]. The recipient runqueue is found via `find_lock_lowest_rq()`, which uses the priority bit array of the root domain (Section 4.2.6.2) to either locate an idle CPU or identify the lowest priority CPU[38] in the system. Once the destination CPU is determined, `pick_next_pushable_task()` selects the task that is finally migrated.

[33] `trigger_load_balance()`, called by `scheduler_tick()`, sets a software interrupt handled by `run_rebalance_domains()`.

[34] If the `SD_LOAD_BALANCE` flag is set for the domain and the balancing interval has expired.

[35] Calculated according to the imbalance

[36] Starting at the domain of the task's CPU and iterating up the hierarchy.

[37] One that has the `SD_LOAD_BALANCE` set.

[38] Out of all busy CPUs, the one that is executing the lower priority task.

`pull_rt_task()` tries to move a task to the local runqueue [Seeker, 2013]. It performs a linear search, through the set of overloaded runqueues referenced by the bitmap of `root_domain` (Section 4.2.6.2), until it finds a source runqueue to push a task from, *i.e.* when that runqueue contains a waiting task of higher priority than those on the local runqueue. If said task is found, it is dequeued from the source runqueue, migrated to the local CPU, and enqueued onto the local runqueue. Note that pull operations are not as precise as push operations, because the source runqueue is not guaranteed to hold the highest priority task out of all waiting tasks in the system.

Regardless of the operation, tasks are only migrated if they are to become the highest priority task on the target CPU. These routines are performed repetitively, on each CPU, with the expectancy of maintaining the system-wide highest priority tasks distributed among the available processors.

Most of the work is done by the `pre_schedule` and `post_schedule` hooks (Section 4.2.4.) of the RT class before and after scheduling decisions. Prior to selecting the next task for execution on a given CPU, `pre_schedule_rt()` calls `pull_rt_task()` to pull a task that can become the highest priority one on the local runqueue. After a context switch, because waiting tasks are unlikely to run on the local CPU in the near future[39], `post_schedule_rt()` calls `push_rt_tasks()` to test if the highest priority waiting task can execute on another CPU.

Another trigger point for `pull_rt_task()` is the `prio_changed` hook of the RT class. Whenever the priority of the executing task decreases, `prio_changed_rt()` must check if a remote task can now become the highest priority task on the local runqueue. Recently awaken tasks may also cause load balancing if the awoken task overloads the runqueue. In this case, the `task_woken` hook of `rt_sched_class` calls `push_rt_tasks()` to try to push a task away.

Naturally, the initial assignment of RT tasks to CPUs is performed by the `select_task_rq` hook of `rt_sched_class` [Seeker, 2013]. On a given CPU, if an awakening task is not to become the highest priority one, `select_task_rq_rt()` tries to find an idle or lower priority CPU for it. If no such unit is found, the task stays on the current CPU and is set to be moved at the nearest opportunity.

4.3 Real-Time Extensions and Related Work

As a GPOS, Linux performs very well with interactive general-purpose applications, but lacks essential features to handle time-sensitive applications correctly and efficiently. In fact, it does not provide a formal means of imposing timing restrictions upon tasks, and is completely alien to the concept of real-time constraints. While the existing scheduling classes and policies in mainline Linux are inadequate for real-time scheduling (as considered in this thesis) other paramount issues such as unbounded execution latency or coarse-grained timing resolution, also hinder system determinism.

Linux was not originally thought as a real-time platform, but its highly customizable nature has always been enticing to the real-time community. Over the years, several companies have

[39] At least not until the next pick round.

developed modified versions of Linux with improved real-time support. Unfortunately, these are often proprietary and restricted to a limited team of collaborators.

Thanks to the remarkable contributions of research teams, scholars, and independent developers, a number of open-source alternatives have also been proposed. While some have experienced great success, and even made it into the kernel mainline, others failed to gain momentum and eventually became obsolete. The following sections briefly describe the most noteworthy efforts.

4.3.1 RTLinux

Real-Time Linux (RTLinux) [RTLinux, Yodaiken, Mar. 1999] is a small and fast RTOS, distributed as patch applicable to the Linux source, that provides Hard Real-Time (HRT) features to the standard Linux kernel.

In practice, the RTLinux kernel makes the guest OS (the standard Linux kernel) fully preemptible by running it as the lowest priority task in the system, while real-time applications execute at a higher priority level without interferences from the guest OS or its process scheduler.

The *Interrupt Abstraction* mechanism of RTLinux offers low-latency interrupt handling capabilities, for real-time applications, in bounded time. It sets up a virtual layer between the guest OS and physical hardware to prevent the guest OS from interfering with the RTLinux kernel and hampering determinism. While hardware interrupts bound to real-time applications are handled immediately by the RTLinux kernel, standard Linux interrupts[40] are dispatched and handled by the guest OS when the RTLinux kernel runs out of work and becomes inactive.

RTLinux employs a Fixed-Priority (FP) scheduling policy (Section 3.3), which schedules the highest priority ready tasks to run first. Although Rate-Monotonic (RM) or Earliest Deadline First (EDF) scheduling can also be activated via loadable kernel modules, real-time scheduling support in RTLinux remains quite elementary.

RTLinux tasks are implemented as loadable modules. These modules run at the highest privilege level, *i.e.* in kernel-space, seriously violating the protected memory space of the kernel. Hence, any errors raised by real-time applications can potentially crash the entire system. Under this approach, real-time applications also cannot use services[41] made available by the guest OS. They are restricted to a minimal set of services provided by RTLinux.

4.3.2 RTAI

Several *spin-offs* of the RTLinux project have emerged to tackle some of its issues. *Real-Time Application Interface (RTAI)* [Mantegazza et al., 2000, RTAI] provides a complete API for real-time tasks, called *Linux-RT (LXRT)*, to overcome the defective framework of RTLinux. RTAI tasks are normal Linux processes running in user-space, but individually bound to a kernel-space *real-time agent* which allows them to transition to *hard real-time mode*. The real-time agent, activated at runtime via system call, has the responsibility of disabling interrupts and assigning

[40] Those associated with non-real-time applications running in the guest OS.
[41] Such as linux device drivers, networking services, *et cetera*.

the task to the RTAI system, moving it from the Linux runqueue to the RTAI scheduler queue. RTAI supports FP and EDF real-time scheduling.

4.3.3 ADEOS, and Xenomai

Adaptative Domain Environment for Operating Systems (Adeos) [Adeos, Yaghmour] consists of a resource virtualization layer between the hardware and the kernel, with the intent of sharing the hardware among several OSs. In Adeos, each OS is represented, and referred to, as a *domain*. Domains are assigned a priority, ordered as such, and set to be notified of specific events[42]. Adeos uses a pipeline of events to propagate them across the configured domains. Events are inserted into the head of the pipeline and travel down to its tail, being either accepted (and handled), stalled, or discarded, by each domain as they progress down.

Xenomai [Gerum, 2002, Xenomai] uses the domain-based interrupt virtualization mechanism of Adeos to execute real-time tasks both in user-space, to avail of the profusion of services offered by standard Linux, and kernel-space, to benefit from HRT execution guarantees. The Xenomai kernel, called *RT-Nucleus*, resides in the *primary domain*, over the *secondary domain* controlled by the Linux scheduler. Tasks start in the primary domain and execute there until they invoke a function belonging to the Linux API, which forces them to migrate to the secondary domain. Once in the secondary domain, a task is handled by the RT class of the Linux scheduler (Section 4.2.4.2) where it can experience some delay or latency[43]. When the function completes, the task returns to the primary domain. Like RTAI and Adeos, Xenomai scheduling is restricted to FP and EDF.

All of the solutions presented in this Section focus on bringing HRT determinism to Linux systems which, although necessary in critical developments, incurs into aggravated execution latency, and tends to underuse the available resources as the criticality of real-time loses significance. Real-time scheduling is rudimentary and primarily directed towards uniprocessor models, performing poorly in the prevalent multi-core platforms of today.

4.3.4 OCERA and AQuoSA

Open Components for Embedded Real-Time Applications (*OCERA*) [OCERA Project] is an open-source project, started in April 2002, and aiming at a POSIX-compliant execution environment for embedded applications. OCERA researchers undertook the development of various real-time features for Linux v2.4 and RTLinux, including a set of real-time schedulers, supporting bandwidth reservation[44] and reclamation, and designed as loadable kernel modules[45]. Prior to module integration, a small patch must be applied to adjust the core scheduler and provide a set of function pointers used by OCERA modules to handle relevant scheduling events. Though simple and flexible, this approach lacked in portability to newer versions of the kernel.

[42] Such as interrupts, system call invocations, task completions, *et cetera*.

[43] Xenomai employs several methods to overcome this. For example, it uses a special domain, the *interrupt shield* to prevent the secondary domain from being interrupted while it is executing a real-time task.

[44] Using the CBS abstraction.

[45] Not to be confused with scheduling modules or classes as presented in Section 4.2.4.

Members of the team then moved onto *Adaptive Quality of Service Architecture* (*AQuoSA*) [AQuoSA Project], an open-source project based on the achievements of OCERA. With the goal of providing Quality of Service (QoS) management capabilities to Soft Real-Time (SRT) applications in Linux, it features a bandwidth reservation and reclamation scheduler following the modular strategy originally designed for OCERA. In essence, AQuoSA ports the main features of OCERA to Linux v2.6, and introduces a few additional components such as a user-space library for feedback-based scheduling.

Though presenting a flexible view of how loadable real-time features can be incorporated into the Linux kernel, none of these solutions are compatible with up-to-date versions of the Linux kernel (and scheduler), nor do they directly support and exploit intra-task parallelism.

4.3.5 PREEMPT_RT

Concisely, the `CONFIG_PREEMPT_RT` patch-set [PREEMPT_RT], originally developed by Ingo Molnar, greatly reduces many sources of latency and unpredictability in the Linux kernel in to make it more determistic.

Hardware interrupts are generated at unpredictable instants, causing a context switch to the respective Interrupt Service Routine (ISR) on a given CPU. PREEMPT_RT, converts ISRs into schedulable and preemptibe kernel threads executing in *process context*[46], as they would otherwise run in non-preemptible *interrupt context*.

In multiprocessor platforms, concurrent kernel threads must serialize to access shared data. To achieve an almost fully preemptible kernel, PREEMPT_RT replaces most in-kernel locking primitives (such as spinlocks) with sleeping RT mutexes supporting *Priority Inheritance (PI)*.

Taking on the work carried out by Thomas Gleixner on kernel timers, PREEMPT_RT also introduces *high-resolution timers* (hrtimers) with nanosecond resolution.

PREEMPT_RT elements such as hrtimers have progressively been incorporated into the mainline, but as many of its features augment determinism at the expense of system overhead and lower throughput, they contradict the main goals of a GPOS. For special purposes, however, it seems to be widely accepted among Linux experts, including Linus Torvalds himself.

Despite the remarkable effort, PREEMPT_RT does not turn Linux into a RTOS[47]. However, valuing enhanced determinism over responsiveness and interactivity, we developed our work in a modified version of the Linux kernel equipped with PREEMPT_RT.

4.3.6 LITMUS^{RT}

LITMUS^{RT} is a plugin-based multiprocessor scheduling extension for the Linux kernel[48]. Since 2006, it has been actively maintained to keep up with recent versions of the kernel[49].

LITMUS^{RT} implements a wide range of scheduling policies for the sporadic SRT task model: from Partitioned, Global, and Clustered EDF, to Partitioned FP, and PD^2. It intends to serve as

[46] Unless explicitly required.
[47] A fully-fledged deterministic OS.
[48] LITMUS^{RT} stands for *Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*.
[49] At the time of this writing, the current version LITMUS^{RT} 2014.2 is designed for Linux v3.10.41.

a useful experimental platform for real-time research, as well as a proof of concept for real-time scheduling theory applied to current hardware. Aspects such as stability, POSIX-compliance, and coding standards, are slightly overlooked, as LITMUS^{RT} does not aspire to become a commercial project, or even merged into the kernel mainline.

Unfortunately, the current LITMUS^{RT} version is restricted to Intel x86-32 and ARM architectures, and neither of the provided scheduling policies enforce bandwidth isolation, nor exploit parallelism in real-time applications.

4.3.7 SCHED_DEADLINE

The *Deadline Scheduling Class*, or *SCHED_DEADLINE*, is a scheduling class for the Linux kernel which implements the M-CBS real-time scheduling policy [Faggioli et al., 2009].

The SCHED_DEADLINE team have experienced great success in November 2013, with its long-awaited inclusion into the mainline kernel v3.14[50].

SCHED_DEADLINE implements a semi-partitioned form of M-CBS, in the sense that tasks are kept in local CPU runqueues, instead of a system-wide queue. By way of CPU affinities, tasks can be assigned to specific CPUs in order to simulate a partitioned approach. Global scheduling is abstracted with an active load-balancing strategy based on push and pull operations between CPUs, much like in the RT class (Section 4.2.4.2).

Each task in the system is bound to exactly one CBS server at all times. Following CBS stipulations (Sections 3.3), each server is characterized by a budget C_i and period T_i, and each task is allowed to execute for (at most) C_i units of time in each period T_i. With this bandwidth reservation technique, SCHED_DEADLINE is able to simultaneously handle periodic, sporadic, and aperiodic tasks, ensuring that they do not interfere with each other in the case of eventual overruns.

SCHED_DEADLINE is outlined in `kernel/sched/deadline.c`[51], with its instance `dl_sched_class` of the `sched_class` data structure. It takes the second place in the scheduling class hierarchy, preceded by `stop_sched_class` and followed by `rt_sched_class`, as shown by Figure .

4.3.7.1 Task Management

SCHED_DEADLINE scheduling entities are defined with the `sched_dl_entity` structure of `include/linux/sched.h`, which extends the process descriptor `task_struct`. It contains the necessary attributes to schedule individual entities such as the absolute and relative deadlines, period, actual and maximum budget[52], or a `rb_node` element to place it in the runqueue.

[50] Linux kernel v3.14 was released halfway through the development of our project. Although we believe that most of our code can be ported to the most recent kernel versions without the need for major adjustments, we have stuck to Linux v3.8.13 until completion.

[51] From Linux v3.14 onwards.

[52] By *actual* budget we refer to the amount left to be consumed at a given time, and by *maximum* budget the value assigned to the CBS server at configuration time

`dl_rq` extends the `rq` data structure with a specific runqueue design for SCHED_DEADLINE. Tasks are maintained in a red-black tree sorted by absolute deadlines, and set up by the `rb_root` attribute of `dl_rq`. Entities are inserted and removed from the runqueue with the `enqueue_task` and `dequeue_task` hooks of the scheduling class. The field `rb_leftmost` of `dl_rq` holds a reference to the leftmost element of the red-black tree, which in this case refers to the highest priority task.

4.3.7.2 Load-Balancing and the cpudl Data Structure

Much like in `rt_sched_class`, in SCHED_DEADLINE most of the load-balancing work is performed by the `pre_schedule` and `post_schedule` hooks of the scheduling class, which continuously reassign tasks to CPUs using a pull and push strategy.

Load-balancing efficiency depends on how quickly the pair of CPUs involved in a push or pull operation is determined. Lacking centralized information about the workload on each CPU, the linear cost of traversing all units in the system can be significant, and imposes a serious scalability bottleneck as the number of CPUs grows. Furthermore, as the system load increases and idle processors become scarce, finding a suitable CPU for a push operation gets ever more difficult[53].

To track the priority of each CPU and identify load-balancing candidates in constant $O(1)$ time, SCHED_DEADLINE uses a binary max-heap, implemented as a two-dimensional bitmap of fixed size, specified by the `cpudl` data structure.

```
1  struct array_item {
     u64 dl;
3    int cpu;
   };
5
   struct cpudl {
7    raw_spinlock_t lock;
     int size;
9    int cpu_to_idx[NR_CPUS];
     struct array_item elements[NR_CPUS];
11   cpumask_var_t free_cpus;
   };
```

Listing 4.5: The `cpudl` data structure.

The binary max-heap, declared as `elements`, is an array of `array_item` objects sorted by `deadline` value of each instance (the *key*). `size` counts the number of CPUs currently referenced by `elements`, while `cpu_to_idx` caches the position of each CPU on the array. A bitmap defined as `free_cpu` maintains a mask of CPUs that are currently not referenced by `elements`.

The structure of `elements` conforms to a set of properties which specify the relationships between nodes and enable quick selection of the nodes with highest and lowest key values. We highlight the following:

[53] Tasks are pushed either to an idle CPU, or to the lowest priority CPU in the system.

- `elements[0]` holds the root of the binary tree, corresponding to the element with the greatest `deadline` value.

- `elements[2*i]` holds the left child node of `elements[i]`, as long as it exists (`2*i` $\leq$ `size`).

- `elements[2*i+1]` holds the right child node of `elements[i]`, as long as it exists (`2*i` $\leq$ `size`).

- `elements[i/2]` holds the parent node of `elements[i]`, as long as it exists (`i` $>$ 0).

- `elements[size-1]` holds the element with the lowest `deadline` value.

A single instance of `cpudl` is declared in `root_domain`, and protected by a spinlock against concurrent changes, performed by each CPU whenever its priority is updated[54]. While this inevitably introduces runtime overhead, it does not compromise the gains in load-balancing efficiency nor the advantages of a semi-partitioned design, as lock contention on this `cpudl` instance is still considerably lower than on a global runqueue of tasks.

`push_dl_tasks()`, called from `post_schedule_dl()`, invokes `push_dl_task()` to try to push a waiting task[55] off the local runqueue. The target CPU for a *pushable task*[56] is determined via call to `cpudl_find()`, coded in `kernel/sched/cpudl.c`. `cpudl_find()` first looks at the `free_cpus` mask of `cpudl` to test (via bitwise intersection) if any idle CPU can receive the task. Alternatively, it checks if the lowest priority CPU, referenced by `elements[0]` of `cpudl`, has a later deadline than that of the pushable task. Queries on `cpudl` via `cpudl_find()` are lock-free. Unfortunately, due to the lack of synchronization between CPUs the system may experience load changes during load-balancing decisions. Hence, these decisions may not be optimal.

`pull_dl_task()`, called from `pre_schedule_dl()`, takes a simpler approach. It iterates through the set of overloaded CPUs, referenced by the bit mask `dlo_mask` of `root_domain`, to find an earlier deadline *pullable task*[57] that can become the highest priority task on the local runqueue. Once such task is found it is pulled to the local runqueue and the pulling operation completes. Note that this mechanism does not deliberately select the highest priority task out of all pullable tasks in the system. Therefore, it is likely to incur priority inversion.

The `cpudl` structure is defined and implemented in `kernel/sched/cpudl.h` and `kernel/sched/cpudl.c`[58]. For more information on the internals and methods of the `cpudl` structure, we refer the reader to the annexed bibliography [Lelli et al., 2011].

[54] When the leftmost element of its local runqueue changes.

[55] All but the leftmost task.

[56] A lower priority task on the local runqueue that is set to be pushed to another CPU.

[57] The second highest priority task on the source CPU.

[58] Replaced by `kernel/sched/cpudeadline.h` and `kernel/sched/cpudeadline.c` from Linux v3.14 onwards.

4.3.8 SCHED_RTWS

To prove their algorithm in real settings, Fonseca [2012] also proposed an implementation of the Real-Time Work-Stealing (RTWS) scheduler (Section 3.3.3.1) as a new scheduling class for the Linux kernel v2.6.36, under the definition of `rtws_sched_class`.

To the extent of our knowledge, and until the publication of this thesis, `rtws_sched_class` is the first and only attempt at combining Global Earliest Deadline First (G-EDF) scheduling with a Work-Stealing (WS) mechanism to exploit intra-task parallelism in real-time applications [59]. Even more remarkably, they were the first to tackle intra-task parallelism and real-time scheduling in Linux with great results. SCHED_RTWS outperformed SCHED_DEADLINE in almost every experimental test, undoubtedly proving the benefits of a WS strategy to appropriately exploit parallelism in real-time applications, thus improving the use of multiprocessing capacity.

As a new scheduling module, `rtws_sched_class` is an instance of the `sched_class` data structure, defined in its own source file `kernel/sched/sched_rtws.c`.

4.3.8.1 Task Management

RTWS entities are represented by the `sched_rtws_entity` structure in `include/linux/sched.h`, instantiated inside the `task_struct`.

Runnable RTWS entities, or tasks[60], are kept either in a global runqueue, or local per-CPU runqueues, in a semi-partitioned form of G-EDF.

The global runqueue is implemented as a red-black tree sorted by non-decreasing order of absolute dealines, outlined by the `global_rq` data structure of `kernel/sched/sched.h`. A single instance of `global_rq` is contended by all CPUs in the system, to insert, remove, and select runnable RTWS tasks for execution.

`rtws_rq` extends the default runqueue of the Linux scheduler with an elaborate implementation of the priority queue of WS deques, described in Section 3.3.3.1. It is based upon two red-black trees sorted by increasing absolute deadlines, namely `pjobs` and `stealable_pjobs`. `pjobs` ties are broken by Last-In First-Out (LIFO), and its leftmost node stores the bottom-most task of the highest priority deque in the local runqueue, *i.e.* the next task to be selected for execution by the `pick_next_task` hook. On the other hand, `stealable_pjobs` ties are broken by FIFO, and its leftmost node holds the topmost thread in the highest priority deque, *i.e.* the next thread eligible for WS.

4.3.8.2 Task Placement

The periodic behavior of each RTWS task relies on a high-resolution timer triggered rhythmically to handle its activations. At each release instant, the hrtimer callback `timer_rtws()` invokes the *dispatching agent* coded in `dispatch_rtws()` to select an appropriate runqueue for the respective task.

[59] See Sections 3.3.2.2 and 2.4.2 for more information on G-EDF and WS, respectively.
[60] Group scheduling is disregarded in `rtws_sched_class`.

`dispatch_rtws()` tries to find an idle or lower priority[61] runqueue for the task. If it doesn't succeed, the task is pushed to the global runqueue.

Unlike other tasks, dynamically generated threads do not initially go through the dispatching agent. A newly spawned task inherits the deadline of its parent, and is enqueued directly in the local runqueue of the CPU where it is originated, according to the logic programmed in `enqueue_task_rtws()` (the `enqueue_task` hook of the scheduling class). All tasks are inserted in the `pjobs` tree, from where they are selected for execution in the local CPU. If the task being enqueued is not currently executing, and there are two or more tasks in `pjobs`[62], it is also enqueued in the `stealable_pjobs` tree to become eligible for WS. Obviously, stolen threads are not enqueued into `stealable_pjobs`.

`put_prev_task_rtws()` implements the `put_prev_task` hook of `rtws_sched_class`. It is invoked by `__schedule()`, before `pick_next_task` (Section 4.2.5.2), to sort out the placement of the task that is potentially leaving the processor. If the task has already been removed from the runqueue then `put_prev_task_rtws()` knows it has finished execution, and takes no further action. Otherwise, the task is likely being preempted and must be placed in the correct runqueue. In case of a thread dynamically generated in the local CPU, it is enqueued in the `stealable_pjobs` tree to become eligible for WS. Otherwise, it is dequeued from the local runqueue (the `pjobs` tree), and enqueued in the global runqueue to contend for one of the available CPUs through G-EDF.

4.3.8.3 Picking the Next Task and Stealing Work

`pick_next_task_rtws()` implements the `pick_next_task` hook of the scheduling class, and selects the next task for execution on a given CPU. First, it looks for work in the local runqueue, and returns the leftmost task in the red-black tree `pjobs`. If the local runqueue has no pending work, a call to `pull_task_rtws()` tries to pull a task from the global runqueue. In case the global runqueue is found empty, a stealing operation is attempted.

SCHED_RTWS adapts the `cpudl` structure of SCHED_DEADLINE (Section 4.3.7.2 to easily find the highest priority stealable task in the system, which is then migrated to the local runqueue and selected for execution. This modified version of the original WS algorithm (Section 2.4.2) is designated by Fonseca [2012] as *Priority-Aware Stealing (PAS)*.

4.4 Plenty of Room for Improvement

The diversity of projects presented in Section 4.3 validate the applicability of Linux as a real-time platform. Projects such as PREEMPT_RT (Section 4.3.5) look to bring basic real-time capabilities to the Linux kernel. Others (Sections 4.3.1, 4.3.2, 4.3.3, and 4.3.4), are able to guarantee HRT determinism, but at the expense of invasive and overbearing techniques.

[61] One in which the leftmost task of `pjobs` has a later absolute deadline than the task being enqueued.

[62] Note that it does not make sense to perform WS unless the runqueue is overloaded. This step is taken after the task has already been inserted into `pjobs`. In other words, the task being enqueued may itself overload the runqueue.

The modular architecture of the Linux scheduler allows easy incorporation of independently developed scheduling classes with new CPU management policies. Nevertheless, *state-of-the-art* solutions offering elementary real-time support are only now making their way into the mainstream. The recent promotion of SCHED_DEADLINE to the mainline kernel is an indication of how real-time support for GPOSs is becoming increasingly relevant, and proves that Linux can perform very well as an Open Real-Time (ORT) system to serve interactive applications and SRT programs with sustainable volumes of throughput and QoS. While other experimental projects such as SCHED_RTWS start focusing on performance, finally exploring the potential of parallel processing of real-time tasks on multi-core chips, we strongly believe that most of the work in this area is yet to be undertaken.

Even though SCHED_DEADLINE can take advantage of SMP configurations, it does not handle intra-task parallelism efficiently. Scheduling of sporadic and aperiodic tasks, with variable execution requirements, is assured by the CBS abstraction, which assigns a fraction of the available processing bandwidth to each real-time application, but cannot use processing capacity left over by early completion of SRT executions (Section 3.3). Our implementation is set to take on and resolve both issues.

SCHED_RTWS showed how WS can be used, in conjunction with a G-EDF scheduling policy, to improve the throughput of real-time tasksets with an astute take on intra-task parallelism. Unfortunately, it is restricted to Worst-Case Execution Time (WCET) scheduling of the periodic task model. Through bandwidth isolation and capacity-sharing, our scheduling class handles both periodic, sporadic, and aperiodic executions seamlessly.

Generally speaking, we join the best features of SCHED_DEADLINE and SCHED_RTWS and introduce a few of our own. The work in SCHED_DEADLINE and SCHED_RTWS has proved invaluable throughout the course of this thesis, providing a rich set of solutions for specific hurdles encountered along the way. As described in Chapter , our implementation of the M-CBS scheduler is inspired by SCHED_DEADLINE, but due to several structural differences, we have found it more beneficial to develop our project *from scratch*, rather than over an altered version of SCHED_DEADLINE. Our WS logic, although similar to that of RTWS, is a part of the novel capacity-sharing strategy proposed by Nogueira and Pinho [2012]. As such, the resulting implementation is also unprecedented.

4.5 Summary

Chapter 5

The SCHED_PCSWS Scheduler

"If you can't explain it simply, you don't understand it well enough."

— Albert Einstein

The introductory notes of Chapter 1 outlined the motivation and main contributions of this project. An overview of parallel computing architectures, programming models, and scheduling strategies, followed in Chapter 2. After a thorough review of real-time systems and scheduling algorithms, Chapter 3 described the algorithmic foundation of our implementation, *i.e.* the Parallel Capacity Sharing by Work-Stealing (p-CSWS) scheduler (Section 3.3.3.2). Chapter 4 studied the internals of the Linux kernel related to scheduling, and surveyed related work on Linux real-time support.

This chapter brings all the pieces together, as we extend the real-time features of the Linux kernel with a new scheduling class for dynamic and irregular parallel real-time applications, based on the original proposal of the p-CSWS scheduler [Nogueira and Pinho, 2012]. We follow the success of the SCHED_DEADLINE project, now regarded a viable solution for Soft Real-Time (SRT) scheduling in open environments, and strive to prove that the heuristic exploitation of parallelism proposed by p-CSWS can be applied to practical scenarios and outperform multiprocessor Constant Bandwidth Server (CBS) scheduling in the presence of dynamic parallel applications. We begin with a description of the system model in Section 5.1. Section 5.2 presents the main design choices and data structures. The implementation is detailed in Section 5.3.

5.1 System Model

Throughout the remainder of this document, we address the scheduling of irregular and dynamic *fork/join* parallel SRT tasks, on parallel platforms comprised of m identical processors $p_1, p_2, \ldots, p_m$.

Each task τ_i can generate a virtually infinite number of multithreaded jobs. A multithreaded job $j_{i,j}$ of a task τ_i alternates between sequential and parallel regions, beginning with a sequential region, and entering a parallel region at any given time, when it splits into an arbitrary number of

concurrent threads. Each thread $w_{i,j}^k, 1 \leq k \leq n_{i,j}$, where $n_{i,j}$ denotes the number of threads in $j_{i,j}$, is generated dynamically, with arbitrary length, at any time during the execution of $j_{i,j}$. All threads of a parallel region synchronize at the end, as the job resumes sequential execution. Other parallel regions can originate from a sequential region at any time.

The number of parallel regions, the number of parallel threads within each region, along with their lengths, are not known beforehand and can vary between jobs of the same task. The Worst-Case Execution Time (WCET) of a job $j_{i,j}$ is given by the total WCET of its threads $WCET = \sum_{k=1}^{n_{i,j}} WCET(w_{i,j}^k)$, *i.e.* the maximum amount of time it takes to execute all threads sequentially on the same Central Processing Unit (CPU).

Each task τ_i is associated with a p-CSWS Dedicated Server (DS) S_i characterized by a budget C_i, a period T_i, and a relative deadline D_i. The j^{th} job $j_{i,j}$ of task τ_i appears at release time $r_{i,j}$ and competes for processor time with deadline $d_{i,k}$, derived from S_i under p-CSWS rules (Section 3.3.3.2). When a DS S_i becomes idle without depleting the reserved capacity the unused bandwidth is donated to a new p-CSWS Residual Capacity Server (RCS) S_j^r, which competes for execution with other bandwidth servers in the system to steal threads from the earliest deadline eligible DS S_s. Each thread $w_{i,j}^k$ is always executing in context of a bandwidth server, either its DS or a RCS, decreasing its budget as processing time is consumed.

Schedulability conditions for a set of n DS, with a total utilization of $U_{\prod} = \sum_i^n U_i$ and a maximum server utilization of $u_{\prod} = max_{1 \leq i \leq n}(U_i)$, on a parallel platform $\prod$ comprised of m unit-capacity CPUs, are given by Equation 5.1.

$$\begin{aligned} u_{\prod} &\leq 1 \\ U_{\prod} &\leq m - (m-1)u_{\prod} \end{aligned} \tag{5.1}$$

5.2 Design Choices and Data Structures

The p-CSWS algorithm [Nogueira and Pinho, 2012] (Section 3.3.3.2) provides a provably correct and efficient method of scheduling concurrent Hard Real-Time (HRT) and SRT tasks, onto the same shared resources, without tardiness interferences. Unfortunately, like most research in real-time scheduling, it is based upon a simplistic system model which disregards a number of real-world problematics. In Section 3.3 we have already addressed the impracticability of these models, devised to validate algorithm usefulness, and some of the complications that arise as we move on to real settings.

Furthermore, as stated in Section 4.4, mainline Linux features are subject to unpredictable latencies that make it virtually unfeasible to achieve HRT determinism on any running event or activity. Programming the Linux kernel is intrinsically complex, laborious, and time-consuming. To benefit the overall performance of the system, respect the internal structure and coding conventions of the Linux scheduler, and assure the theoretical correctness of the algorithm, we were compelled to adapt some aspects of the proposed design. These changes are duly justified in the following sections.

5.2.1 Base System and Source Files

SCHED_PCSWS was implemented upon Linux v3.8.13, with the PREEMPT_RT-rt12 (Section 4.3.5) patch applied. To enable or disable SCHED_PCSWS at compilation time, a new entry has been added to Kconfig.

The bulk of our policy is implemented in two new source files. `kernel/sched/pcsws.c` (abbreviated to `pcsws.c`) defines the new scheduling class, along with its methods and auxiliary functions. `kernel/sched/pcsws.h` (`pcsws.h`) specifies the headers for the functions exported by `pcsws.c`.

Our extension of the generic per-processor runqueue was defined in `kernel/sched/sched.h` (`sched/sched.h`). p-CSWS schedulable entities were coded in `include/linux/sched.h` (`linux/sched.h`). Integration with the core scheduler was performed in `kernel/sched/core.c` (`core.c`).

The `cpudl` data structure of SCHED_DEADLINE and its methods were adapted to our needs in `kernel/sched/cpudl.c` (`cpudl.c`) and exported in `kernel/sched/cpudl.h` (`cpudl.h`).

5.2.2 p-CSWS Scheduling Class and Policy

At the heart of SCHED_PCSWS is the `pcsws_sched_class` scheduling class (Listing 5.1), defined along with all scheduling methods in `pcsws.c`.

```
const struct sched_class pcsws_sched_class = {
  .next                 = &rt_sched_class,
  .enqueue_task         = enqueue_task_pcsws,
  .dequeue_task         = dequeue_task_pcsws,

  .check_preempt_curr   = check_preempt_curr_pcsws,

  .pick_next_task       = pick_next_task_pcsws,
  .put_prev_task        = put_prev_task_pcsws,

#ifdef CONFIG_SMP
  .select_task_rq       = select_task_rq_pcsws,
  .migrate_task_rq      = migrate_task_rq_pcsws,

  .pre_schedule         = pre_schedule_pcsws,
  .post_schedule        = post_schedule_pcsws,
  .task_waking          = task_waking_pcsws,
#endif

  .set_curr_task        = set_curr_task_pcsws,
  .task_tick            = task_tick_pcsws,
  .task_fork            = task_fork_pcsws,

  .task_dying           = task_dying_pcsws,
  .task_dead            = task_dead_pcsws,

```

```
    .switched_from          = switched_from_pcsws,
    .switched_to            = switched_to_pcsws,
};
```

Listing 5.1: The p-CSWS Scheduling Class `pcsws_sched_class`.

`sched_pcsws_class` was configured as the second scheduling class on the hierarchy, as depicted by Figure 5.1. Consequently, p-CSWS tasks can only be preempted by the per-CPU stopper task (Section 4.2.4.1), as is essential for the correct functioning of the system.

Figure 5.1: Integrating `pcsws_sched_class` onto the scheduling class hierarchy

Linux tasks are tied to a given scheduling class policy through the `policy` attribute of `task_struct` (Sections 4.2.1 and 4.2.4). Listing 5.2 shows how we have extended the set of policy identifiers (Section 4.2.4) with a new macro, defined as `SCHED_PCSWS`, to identify p-CSWS tasks.

```
[...]
#define SCHED_IDLE          5
#ifdef CONFIG_SCHED_PCSWS_POLICY
#define SCHED_PCSWS         6
#endif
```

Listing 5.2: SCHED_PCSWS policy identifier.

5.2.2.1 Additional Scheduling Methods

To handle task termination events, a pair of scheduling methods were added to the generic `sched_class` structure of `linux/sched.h` (Section 4.2.4), as shown by Listing 5.3.

```
struct sched_class {
[...]
  void (*task_dying) (struct task_struct *p);
  void (*task_dead) (struct rq *rq, struct task_struct *p);
[...]
```

Listing 5.3: Extending the `sched_class` data structure.

- `task_dying` is invoked at the beginning of `do_exit()` in `kernel/exit.c` (Section 4.2.5.3), right after a task enters the dying process.

- `task_dead` hooks the context switch that takes a task off the CPU for the last time before it dies. It is called from `finish_task_switch()` in `kernel/sched/core.c`.

5.2.3 Schedulable Units

To ease discussion, we refer to a parallel p-CSWS task, from which parallel threads are dynamically generated, as a *master task* or *parent task*. p-CSWS threads created by the master task are referred to as *child* tasks or threads.

Because it is considerably simpler to understand, implement, and use, a relationship of one-to-one between DSs and p-CSWS master tasks is in effect. By default, the master task and all of its child threads are served by a specific DS, but they may temporarily be scheduled by a RCS as a result of a Work-Stealing (WS) operation. We define 3 distinct execution contexts for runnable p-CSWS tasks:

- *DS-bound* tasks are non-stolen tasks executing in the context of the a DS. The set of DS-bound tasks served by the same DS is called a *p-CSWS group*, or simply *group*. All tasks of a p-CSWS group are affined to the same CPU, called the *group CPU*. DS-bound tasks are scheduled by the timing constraints of their DS and consume its budget directly.
- *RCS-bound* tasks are stolen tasks executing in the context of a RCS. Executing *RCS-bound* tasks consume the budget of the associated RCS.
- An *unbound task* is a non-stolen task that is temporarily separated from its group, executing on another unit other than its group CPU. Unbound tasks are scheduled by the deadline of their DS, but do not consume its budget directly. The execution time is stored and subtracted from the budget of the DS once the task rejoins its group.

5.2.3.1 p-CSWS Tasks

As mentioned in Section 4.2.1 Linux processes and threads are instances of the process descriptor, *i.e.* the `task_struct` data structure.

We disregard group scheduling (Section 4.2.6.2) and represent each p-CSWS task as an instance of a p-CSWS scheduling entity, defined in `linux/sched.h` as the `sched_pcsws_entity` data structure.

```
1  struct sched_pcsws_entity {
   /* Parent sched_pcsws_entity. (=self for parent tasks) */
3  struct sched_pcsws_entity *parent;

5  struct pcsws_rcs *thief; /* Thief RCS. (NULL when task isn't stolen) */

7  struct pcsws_ds dedicated; /* pCSWS Dedicated Server */
   struct pcsws_rcs residual; /* pCSWS Residual Capacity Server */
9
   struct rb_node boosted_node; /* Boosted rb_tree node */
11 struct list_head deboosted_node; /* Deboosted list node */
   struct list_head ready_deque_node; /* Deque node (for pCSWS threads) */
13 struct list_head stolen_node; /* Deque stolen node (for pCSWS threads) */

15 u64 overrun_amount;
```

```
17   int throttled;
};
```

Listing 5.4: The `sched_pcsws_entity` data structure.

This structure is initialized differently for master tasks and child threads. In entities representing child threads, the pointer `parent` is initialized to reference the parent task. By convention, master tasks reference themselves, as the pointer is set to the enveloping `sched_pcsws_entity` instance.

Stolen tasks are identifiable by the attribute `thief`, which points to the associated RCS.

Given the one-to-one correspondence between servers and parallel tasks, each DS serves a single master task and its dynamically generated child threads. The attribute `dedicated` is initialized exclusively for instances of `sched_pcsws_entity` that represent master tasks, and left unused for instances representing child threads. Instead, threads are associated with the DS set up by the master task, accessible through the `parent` pointer of `sched_pcsws_entity`.

`residual` represents a RCS and is also initialized exclusively for master tasks, as explained in Section 5.3.8

5.2.3.2 Dedicated Servers

DSs are instances of the `pcsws_ds` data structure defined in `linux/sched.h`.

```
struct pcsws_ds {
2   u64 period; /* Server period */
    u64 budget; /* Reserved capacity */
4
    struct ws_deque deque; /* Work-stealing deque */
6
    unsigned long nr_threads; /* Number of child threads currently running */
8
    unsigned long nr_stolen; /* Number of stolen threads */
10  struct list_head stolen_bottom; /* Stolen tasks. */

12  struct sched_stats_pcsws stats; /* Group statistics */
};
```

Listing 5.5: The `pcsws_ds` data structure.

`pcsws_ds` holds the timing parameters needed to characterize a DS, specifically the period and reserved budget.

All runnable DS-bound tasks served by the DS are kept in a WS deque (Section 5.2.3.4) declared as `deque`. `nr_threads` is a counter for the number of active child threads generated by the master task.

`stolen_bottom` sets up the list of stolen tasks described in 3.3.3.2, henceforth referred to as the *stolen list*. p-CSWS tasks are classified as *stolen* as long as they exist on the stolen list of

their DS, and *non-stolen* otherwise[1]. `nr_stolen` counts the number of stolen tasks on the stolen list.

5.2.3.3 Residual Capacity Servers

Choosing how to represent and manage RCSs proved to be one of the most delicate dilemmas encountered in the course of this project.

To represent RCSs a new data structure was defined, as `pcsws_rcs`, in `linux/sched.h`.

```
struct pcsws_rcs {
  struct rb_node node; /* Local RCS rq node */

  struct rq *rq; /* Currently assigned rq */

  struct task_struct *stolen_thread; /* Currently stolen thread */
  struct ws_deque deque; /* work-stealing deque */

  int local; /* Binary flag. 1 if deque->budget <= QRES_MIN */
};
```

Listing 5.6: The `pcsws_rcs` data structure.

As explained in Section 5.3.8, `node` allows enqueueing the RCS onto the per-processor red-black tree of RCSs described in Section 5.2.4, while `rq` points to the currently associated run-queue. `deque` is deque of stolen tasks, which also defines the timing parameters of the RCS (Section 5.2.3.4). `local` is a binary flag which defines whether the RCS can execute on any CPU to steal tasks (0), or must be donated to the local CPU (1) as stipulated by Rule F of the p-CSWS scheduler.

To manage RCS 3 different design choices were contemplated:

1. **Dynamic memory allocation** - Several attempts were made to create `pcsws_rcs` instances dynamically, at job completion. Unfortunately, we have found it extremely troublesome to manage `kmalloc()` and `kfree()` calls from scheduler code, and after much cogitation the idea was abandoned.

2. **Preallocate RCSs** - We would allocate a static buffer of `pcsws_rcs` instances at configuration time, and reuse these instances when needed. Although simple and easily implementable, this alternative was not flexible, as it forced an upper bound on the number of active RCSs and subsequently on the achievable parallelism. For this reason, it was also abandoned.

3. **Create a `pcsws_rcs` instance per DS or master task** - Seeing as each task can only generate a new RCS per job (at response time)[2], we would establish a relationship of one-

[1] Although the notions of stolen and RCS-bound task (Section 5.2.3) may be similar, they are not interchangeable. RCS-bound tasks must be on the deque of their thief RCS, while stolen tasks may already have been taken of the thief deque but still exist on the stolen list of their DS. Section 5.3.2 shows when a returning task is dequeued from the stolen list

[2] We assume that no two jobs $j_{i,a}, j_{i,b}$ of a p-CSWS task τ_i overlap in time. $\forall a, b \in \{1, .., \infty\}, a < b, f_{i,a} \leq r_{i,b}$.

to-one between DSs and RCSs, by assigning a `pcsws_rcs` instance to every DS or master task[3], and simply enqueue that instance onto a queue of active RCSs to mark it as active.

Unfortunately, under the considered task model it is possible that a task τ_i generate overlapping RCS, if a job $j_{i,j+1}$ finishes before the deadline of the preceding job $j_{i,j}$. In which case, there may exist an active RCS S_j^r, generated by $j_{i,j}$, with deadline $d_j^r = d_{i,k}$ at time $t = f_{i,j+1}$, when a new RCS S_{j+1}^r is being generated by $j_{i,j+1}$. If neither S_j^r or S_{j+1}^r deplete their capacities, they coexist in the system until d_j^r, when S_j^r expires and is removed from the system.

According to Rules D and F of the p-CSWS algorithm, a new RCS is generated whenever a DS is becoming idle. Under our model, since each DS is allotted to a single parallel task, and p-CSWS jobs do not interrupt execution voluntarily, a DS becomes idle when the current served job $j_{i,j}$ completes, at response time $t = f_{i,j}$. Thus, a RCS S_j^r is dynamically generated by the k^{th} instance of a DS S_i, at response time $f_{i,j}$, with deadline $d_j^r = d_{i,k}$ and budget $c_j^r = min(c_{i,k}, d_{i,k} - t)$.

For an aperiodic task τ_i no restrictions are made regarding the release time between consecutive jobs. Therefore, it is possible that the DS S_i serving τ generate multiple RCSs $S_j^r, S_{j+1}^r, \ldots, S_n^r$ that overlap in time, if $f_{i,j+1}, f_{i,j+2}, \ldots, f_{i,n} \leq d_j^r$. This scenario is depicted in Figure .

This cannot be fulfilled using a single `pcsws_rcs` instance per master task, as outlined by design choice 3. We have realized that, in order to comply with this requirement, each DS had to create independent `pcsws_rcs` instances dynamically, as in design choice 1, so that various RCSs generated by the same DS could coexist and overlap in time. Although no solid documentation to support legitimate our claim has been found, we have encountered issues invoking `kmalloc()`[4] and `kfree()` from atomic context under `PREEMPT_RT`[5]. Dynamic memory allocation increased the complexity of our implementation dramatically, and raised runtime overhead and unpredictability, always undesirable on a real-time environment. After expending a great amount of time trying to identify and resolve implementation flaws, we eventually gave up this design choice. Seeing as only aperiodic tasks can create overlapping RCSs, in very specific conditions, we decided to pursue design choice 3 and improve this feature at a later date.

A `pcsws_rcs` instance, named `residual`, was embedded into `sched_pcsws_entity`. The variable is initialized only for entities representing master tasks, and left unused for dynamically generated threads. In doing so, we consider that each master task owns both a DS and a RCS and, upon job completion, the `pcsws_rcs` instance owned by the master task can be enqueued onto a queue of active RCS, from which it can be selected for execution and steal work until its capacity is exhausted or expired. To respect bandwidth reservations and cope with the impossibility of creating overlapping RCSs, when a DS becomes active, at job arrival, and finds its associated RCS active but unused, we deactivate the RCS and donate its capacity back to the DS owned by its master task. If the RCS is currently executing a stolen thread it remains active, retaining its capacity, and a new DS instance is created to execute the new job. Once a job finishes,

[3]Recall that master tasks and DSs exist in a relationship of one-to-one, therefore it is irrelevant to associate each the RCS with a DS or with the corresponding master task.

[4]Even with the `GFP_ATOMIC` flag set.

[5]The scheduler runs almost entirely in atomic context, with preemption disabled.

the DS checks if its sibling[6] RCS is active. If so, it is deactivated and reactivated, with new timing parameters, as a new RCS instance. A more detailed explanation of how RCSs are managed by `SCHED_PCSWS` is presented in Section 5.3.8.

By forcing the deactivation of a RCSs that might have a chance to steal tasks in the future, the measures presented above may cause a slight decrease on the overall parallelism. However, if the an RCS S_j^r generated by S^i remains unused at arrival of the next job served by S^i, it means that either (i) no task is eligible to be stolen by S_j^r, or (ii) the deadline d_j^r is too far to bring S_j^r to execution. In which case, unless an eligible task appears in the system before deadline d_j^r, S_j^r will remain unused and waste its capacity.

5.2.3.4 Work-Stealing Deques

WS deques are instances of the `ws_deque` data structure presented in Listing 5.7.

```
/*
* Updates must hold cpu_rq(ws_deque.cpu)->lock
*/
struct ws_deque {
  int cpu; /* Currently assigned cpu */

  int dedicated; /* 1 for DS deque, 0 for RCS deque */

  unsigned long nr_running; /* Number of non-stolen threads in the deque */
  unsigned long nr_ready; /* Number of non-stolen threads in the deque */

  struct list_head ready_bottom; /* Work-stealing deque ready tasks */
  struct list_head exiting_bottom; /* Work-stealing deque ready tasks */

  struct rb_node local_node; /* Local rq node. */

  s64 budget; /* Current budget */
  u64 deadline; /* Absolute deadline */

  int help_first; /* Chosen work-stealing policy. 1 for help-first, 0 for work
      -first */
};
```

Listing 5.7: The `ws_deque` data structure.

Each deque is always associated to a certain CPU, specified by the `cpu` variable of `ws_deque` and termed *group CPU*. Since most task manipulations and scheduling operations must be performed with the local runqueue lock held, we avoid complex locking rules and protect each deque with the lock of the runqueue where it currently resides.

Seeing as all tasks on a deque share the same timing constraints, the `ws_deque` element itself holds the current budget and absolute deadline (defined as `budget` and `deadline`) inherited by all served tasks. Each p-CSWS task is scheduled according to the timing constraints of its *current*

[6]The RCS represented by the `pcsws_rcs` belonging to the encapsulating `sched_pcsws_entity`.

deque. The current deque of a RCS-bound task is the deque of its thief RCS, whereas the current deque for other tasks is the deque of their DS[7].

The binary flag `dedicated` denotes whether the encapsulating data structure is of type `pcsws_ds` (`1`) or `pcsws_rcs` (`0`). The first case is referred to as a *dedicated deque*, while the latter is termed *residual capacity deque*.

`ready_bottom` defines the actual deque of runnable tasks served by the encapsulating server.

Two WS sub-policies have been proposed and studied in literature [Guo et al., 2009]. In *work-first*, as considered by Nogueira and Pinho [2012], the newly created thread is immediately attributed to the CPU after a fork operation, leaving the master task as the only task eligible for WS and providing a greater control on the generated parallelism[8]. In *help-first*, the master task keeps executing after each fork statement, and spawning threads that will become available for WS[9]. Seeing as tasks are traditionally selected for execution on the local CPU from the bottom of a deque, the difference between a help-first and work-first strategy depends on whether the master task is always enqueued at the bottom of the deque or not. As explained in Section 5.3.2 we have decided to implement both sub-policies, as that only requires a slight adaptation of the enqueue mechanism.

5.2.4 Runqueues

In Section 3.3.2.1 we have seen how the efficiency and correctness of multiprocessor real-time schedulers are contingent upon fundamental design choices, such as the organization of tasks under a global, partitioned, or semi-partitioned scheme. The description in Nogueira and Pinho [2012] suggests a semi-partitioned configuration, where a global queue of active bandwidth servers is shared among processors, and local processor queues are used to store runnable tasks executing in the context of busy p-CSWS servers.

Unfortunately, due to lock contention on the global queue, this approach is not scalable as the overall performance of the system is bound to degrade with an increasing number of CPUs. In conformity with the structural design of the Linux Scheduler, we replace the global queue of active servers with a more scalable semi-partitioned design, paired with an active load-balancing scheme that seeks to preserve priority correctness and maximize resource utilization across the multiprocessor platform.

At a first stage we schedule deques onto processors, rather than individual tasks. The `rq` structure is extended with our own specification of a per-processor runqueue for `pcsws_sched_class`, the `pcsws_rq` data structure declared in `sched/sched.h`.

```
1 struct pcsws_rq {
  unsigned long nr_running; /* deques. */
```

[7] `task_curr_deque()` returns the current deque of a p-CSWS task passed as a parameter. Although unbound tasks do not execute in the context of their DS, they share the same absolute deadline as their group.

[8] Under work-first, parallelism is generated when strictly necessary, *i.e.* the master task can only generate multiple parallel threads if another CPU succeeds at stealing it.

[9] In this explanation we consider a basic fork/join model without nested parallelism.

```
3   struct rb_root deques; /* Local tree of active deques, sorted by (abs)
        deadline */

5   struct rb_node *leftmost; /* Leftmost deque on the rb-tree. */
    u64 earliest_dl; /* Earliest deadline out of all deques on the rq. */
7
    struct rb_node *pushable; /* Earlist pushable deque. */
9   u64 earliest_pushable_dl; /* Earlist pushable deque (abs) deadline. */

11  struct ws_deque *stealable; /* Stealable deque. */

13  struct pcsws_rcs_rq residual; /* RCSs currently assigned to the rq. */

15  struct pcsws_boosted boosted; /* Boosted tasks. */

17  /* Stats */
    unsigned long tot_migrations;
19  unsigned long tot_switches;
    unsigned long tot_steals;
21 };
```

Listing 5.8: The `pcsws_rq` data structure.

At the heart of `pcsws_rq` is a red-black tree of `ws_deque` instances, defined as `deques`, with size `nr_running`. At a given point in time, `deques` stores all non-empty deques assigned to the local CPU, sorted by increasing order of absolute deadlines.

A reference to the leftmost node of the tree is kept in `leftmost`, while the absolute deadline of the corresponding deque, called *leftmost deque*, is cached in `earliest_dl`. As the earliest deadline out of all deques on the local runqueue, `earliest_dl` defines the priority of the local CPU.

For load-balancing purposes, `pushable` and `earliest_pushable_dl` keep references to the *pushable deque*, *i.e.* the second highest priority deque on the runqueue, and its absolute deadline. The role of these attributes will be detailed in Section 5.3.5.

`stealable` points to the leftmost deque of the tree when it is eligible for WS, i.e. when the leftmost deque is a dedicated deque holding two or more runnable tasks. In this case we refer to the leftmost deque as *stealable deque*.

`tot_migrations`, `tot_switches`, and `tot_steals`, are statistic counters to the number of migrations, context switches, and WS operations, involving p-CSWS tasks on the local CPU.

5.2.4.1 The RCS Runqueue

The theoretical proposal states that active RCSs are released to the global queue of active servers, to compete for CPU time with other bandwidth servers in the system. However, a RCS can only be selected for execution if its deque is not empty, *i.e.* if it succeeds at stealing a thread from a DS eligible for WS.

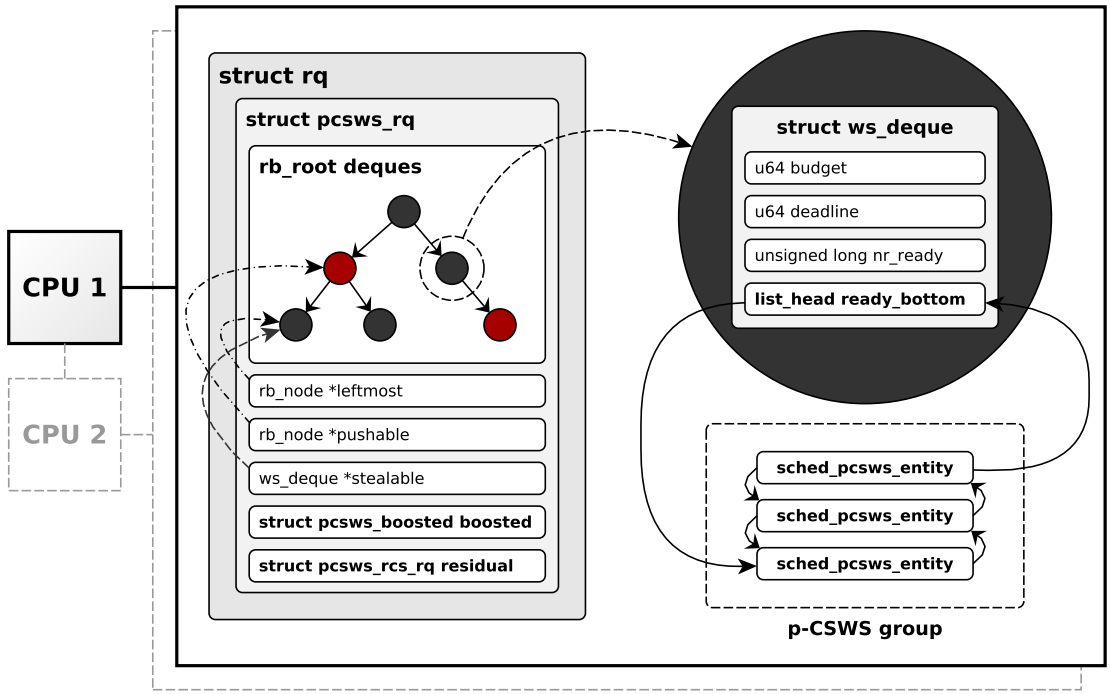

Figure 5.2: Overview of the SCHED_PCSWS runqueue design

If active DSs and RCSs compete for execution on the same priority queue, then this queue may hold empty RCS which complicate scheduling decisions. For instance, to select the highest priority non-empty server for execution, we either waste computational cycles traversing the queue, or employ additional programming logic to cache its position on the tree.

Instead, we organize active RCS separately, onto per-CPU extensions of the p-CSWS ruqueue called *RCS runqueues*. RCS runqueues are instances of the `pcsws_rcs_rq` structure defined in `sched/sched.h`. Each p-CSWS runqueue `pcsws_rq` is extended with an instance of `pcsws_rcs_rq`, named `residual`, to set up a per-CPU RCS runqueue.

```
struct pcsws_rcs_rq {
  unsigned long nr_running; /* Pushable RCS counter */
  struct rb_root rcs_root; /* Pushable RCS tree */
  struct rb_node *leftmost; /* Pushable RCS leftmost */
  u64 earliest_dl; /* Pushable RCS earliest deadline */

  unsigned long local_nr_running; /* Local RCS counter */
  struct rb_root local_rcs_root; /* Local RCS tree */
  struct rb_node *local_leftmost; /* Local RCS leftmost */
  u64 local_earliest_dl; /* Local RCS earliest deadline */

  struct pcsws_rcs *idling_rcs; /* Idling RCS */
  u64 idling_start; /* Idling timestamp */
  struct hrtimer idling_timer; /* Idling timer */
};
```

Listing 5.9: The `pcsws_rcs_rq` data structure.

Two red-black trees, and auxiliary variables, are specified in `pcsws_rcs_rq`. To materialize Rule E of the p-CSWS scheduler, we have decided to organize RCSs with a capacity lower than a pre-established value `Q_MIN`[10] on a red-black tree of their own, defined as `local_rcs_root`. These `pcsws_rcs` instances, which we designate *local RCSs* are donated to next task executing on the local CPU, and cannot steal work.

RCSs with a capacity greater than `Q_MIN` are stored in `rcs_root`. These servers, henceforth referred to as *pushable RCSs*, can move between CPU to steal work and compete for execution with other bandwidth servers in the system.

The auxiliary attributes for `rcs_root` and `local_rcs_root` are self-explanatory.

According to Rule I of the p-CSWS algorithm, when the local runqueue becomes idle the earliest deadline waiting pushable RCS is assigned to the local CPU via the `idling_rcs` pointer, and its capacity is decreased as if it were executing a p-CSWS task. `idling_start` is a timestamp for the last budget update. `idling_timer` is a hrtimer set to fire at RCS exhaustion or expiration time.

Section 5.3.8 provides an in-depth explanation of how RCSs and RCS runqueues are managed. Section 5.3.9 describes the implemented WS mechanism.

5.2.4.2 The Boosted Runqueue

Our distributed approach relies on a dynamic load-balancing strategy that continuously redistributes active tasks to CPUs. For several reasons the kernel often changes the affinity of a task, binding it to its current CPU and forbidding it to migrate. If not properly handled, this interference can seriously compromise our ability to maintain the highest priority task executing on the available CPUs.

Consider the following example of 2 tasks executing on a parallel system of 2 CPUs. During a fork operation the kernel pins the master task τ_a to CPU 1 until the fork operation completes and newly created thread is ready to execute. During this process, a higher priority p-CSWS task τ_b, which is also pinned to CPU 1, wakes up, preempts τ_a, and starts executing straight away. In the meanwhile, CPU 2 remains idle, and τ_a stays pinned to CPU 1 until it can execute again and complete the fork. This is a severe case of priority inversion which can also lead to an unpredictable and dramatic decrease in resource utilization.

To resolve this, we occasionally *boost* the execution of a p-CSWS task, allowing it to overtake the CPU and quickly resolve an impediment that may be binding it to the current CPU. Boosted tasks preempt other local p-CSWS entities, and execute at the highest priority for a minimum amount of time. Even though we may be forcing priority inversion on the local CPU for very short periods of time, task boosting is used purely as a last resort measure, in cases where it allows us to avoid larger and uncertain periods of real-time incorrectness.

A task enters the boosted state when it is enqueued onto the local runqueue of boosted tasks. This runqueue is an instance of the the `pcsws_boosted` structure, defined as `boosted` in `pcsws_rq`.

[10] `Q_MIN` is defined as a macro in `pcsws.c`.

```
struct pcsws_boosted {
  unsigned long nr_running; /* Boosted tasks counter */
  struct rb_root tasks; /* Boosted tasks */

  struct rb_node *leftmost; /* Leftmost task on the boosted tree */

  unsigned long nr_deboosted; /* Deboosted tasks counter. */
  struct list_head deboosted; /* Deboosted tasks. */
};
```

Listing 5.10: The `pcsws_boosted` data structure.

`pcsws_boosted` holds a red-black tree (`tasks`) of individual tasks sorted by increasing absolute deadlines. At the first scheduler tick, a boosted task running on the local CPU is dequeued from `task` and enqueued onto the Last-In First-Out (LIFO) list `deboosted`, from which it is dispatched to the correct CPU.

A thorough description of the task boosting mechanism will be presented in Section 5.3.4.

5.2.5 Global Scheduling Data

p-CSWS rules impose that, at any given point in time, the n highest priority tasks be allotted to the m available CPUs ($n \leq m$), in Global Earliest Deadline First (G-EDF) fashion. While this requirement is met straightforwardly using a single task queue, as we switch to a decentralized assignment of schedulable elements to CPUs a fast and complete view of these mappings is essential to make quick and accurate global scheduling decisions.

The `cpudl` data structure implemented by SCHED_DEADLINE, and described in Section 4.3.7.2, provides a centralized way of tracking load dynamics on each CPU at a low algorithmic and contention overhead. To support load-balancing and WS decisions across the domain, we have decided to extend the `root_domain` data structure (Section 4.2.6.2) with 4 instances of `cpudl` as follows:

- `pcswsc_cpudl` tracks the priority of each busy[11] CPU.
- `pcswso_cpudl` tracks CPUs with pushable deques on their local runqueues and the priorities of these pushable deques.
- `pcswsr_cpudl` tracks the highest priority pushable RCS on each CPU.
- `pcswss_cpudl` tracks CPUs with stealable deques, *i.e.* when the leftmost deque is a dedicated deque holding more than one task.

As explained in Section 5.3.2, `pcswsc_cpudl` and `pcswso_cpudl` are updated by each CPU whenever their leftmost and pushable deques change, respectively.

`pcswsr_cpudl` is part of the implemented WS mechanism. Its role will be explained in Sections 5.3.8 and 5.3.9.

[11] Holding at least 1 deque on its local p-CSWS runqueue.

pcswss_cpudl allows quick selection of WS victims. As described in Section 5.3.2, it is updated whenever the local stealable deque changes, and consulted by the WS mechanism presented in Section 5.3.9.

5.3 Implementation

Having seen how pcsws_sched_class was defined, and how schedulable p-CSWS units are represented and organized, we now describe how these units are managed and scheduled for execution. Most features of SCHED_PCSWS are implemented as methods of the pcsws_sched_class, following the modular structure of the Linux scheduler.

5.3.1 Launching Tasks and Threads

At creation time, p-CSWS tasks are regular Linux tasks spawned using the fork() and clone() family of system calls. Tasks can change their scheduling policy through the sched_setscheduler() system call, passing the new policy identifier as a parameter.

Introducing a new p-CSWS to the system implies specifying a new bandwidth reservation under the form of a p-CSWS DS. When a task changes its policy to SCHED_PCSWS, it must define the reserved amount through a pair of values for the budget and period of the DS. Because this feature is not permitted by the default sched_setscheduler() system call, we require user processes to transition to SCHED_PCSWS using a new system call defined in kernel/sched/core.c as sched_setscheduler_pcsws().

The actual policy switch is performed by __sched_setscheduler() of kernel/sched/core.c. We have adapted __sched_setscheduler() to allow initialization of p-CSWS master tasks, according to the parameters passed from the sched_setscheduler_pcsws() system call. If the calling task is switching to SCHED_PCSWS, passing the identifier 6 to the policy parameter of sched_setscheduler_pcsws(), a call to __setparam_pcsws() initializes the sched_pcsws_entity of the process descriptor. init_pcsws_ds() configures the DS with the budget and deadline passed from user-space. init_pcsws_rcs() sets up the RCS owned by the master task. init_pcsws_parent() initializes the scheduling entity and sets up the first server instance via replenish_ds(). Back in __sched_setscheduler(), once the task is enqueued onto the local p-CSWS runqueue, it is managed and scheduled for execution as a p-CSWS task.

Child p-CSWS threads are generated via traditional forking of p-CSWS tasks at runtime. Forking events are handled by the sched_fork() routine of kernel/sched/core.c. We have adapted sched_fork() to set the parent pointer of the new sched_pcsws_entity instance, before class-specific configurations are delegated to the task_fork method of pcsws_sched_class. In task_fork_pcsws() the nr_threads counter of the parent DS (Section 5.2.3.2) is incremented and the remainder attributes of the sched_pcsws_entity descriptor are initialized via init_pcsws_thread(). Because child threads inherit the timing constraints imposed by the DS owned by their parent and cannot share residual capacities, the attributes dedicated and

`residual` of their `sched_pcsws_entity` are initialized with ineffectual data and remain inoperative throughout execution. Once a parallel thread is configured, it is mapped to the group CPU (Section 5.3.3) and activated via `wake_up_new_task()` (Section 4.2.5.3).

5.3.2 Activating and Deactivating Tasks

When a p-CSWS task switches to the `TASK_RUNNING` state `try_to_wake_up()` calls `activate_task()` to enqueue it onto a runqueue of runnable tasks. On the other hand, as a task switches to a non-runnable state, `deactivate_task()` removes the task from the list of runnable tasks so that it cannot be picked for execution. Seeing as runqueue implementation differs between scheduling classes, these operations are delegated to class-specific functions through the `enqueue_task` and `dequeue_task` hooks of the scheduling class, respectively.

The `enqueue_task` method of `pcsws_sched_class` is implemented by `enqueue_task_pcsws()` (Figure 5.3) in `pcsws.c`. In Section 5.2.3 we have mentioned that all tasks on a p-CSWS group must be affined to the same CPU, Here, we allow tasks to be enqueued onto a WS deque only if they are RCS-bound or on the correct CPU. If the task is found to be on the wrong CPU via `wrong_cpu()`, it is enqueued onto the boosted runqueue to rejoin its group on the next invocation of the main scheduler (Section 5.3.4).

Otherwise, we may be dealing with a stolen task that is returning to its group. If the task is on the stolen list of its DS, we dequeue it from that list and check if any unbounded execution time needs to be accounted (Section 5.3.10).

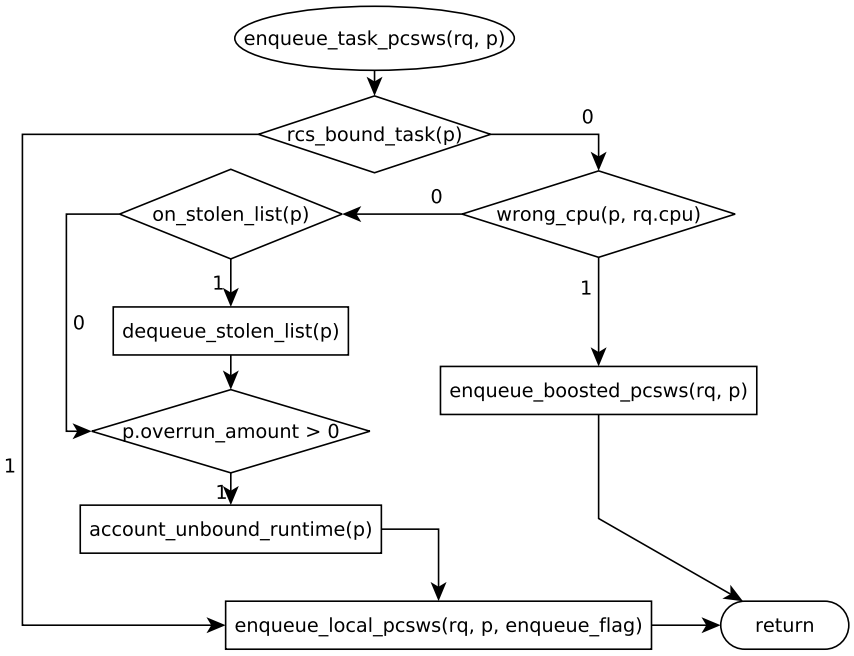

Figure 5.3: Control flow diagram for `enqueue_task_pcsws()`

A call to `enqueue_local_pcsws()` (Figure 5.4) effectuates the enqueue. First, `enqueue_ready_deque()` inserts the task onto its *current deque* (Section 5.2.3.4) determined by the `task_curr_deque()` function of `pcsws.c`. Stolen tasks are always enqueued at the bottom of the deque, while in-

sertion of DS-bound tasks depends on the chosen WS policy for the deque (Section 5.3.2). If the `help_first` attribute of `ws_deque` is unset, we simply enqueue the task at the bottom of the deque. Otherwise, we check if the task being enqueued is a master task that should be enqueued at the bottom. Threads are either enqueued at the bottom or at the second bottom-most position of the deque, depending on whether the master task is on the deque or not. Note that regardless of the position where a task is enqueued, this operation is always subject to $O(1)$ complexity.

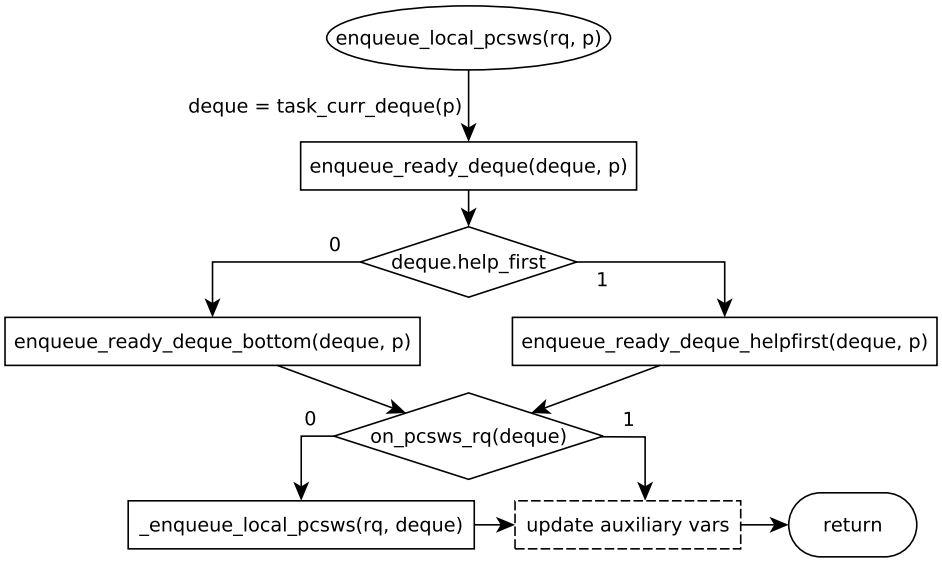

Figure 5.4: Control flow diagram for `enqueue_local_pcsws()`

Back in `enqueue_local_pcsws()`, if the deque is not yet on the local runqueue it is enqueued onto the `deques` tree of `pcsws_rq`[12], following a call to `_enqueue_local_pcsws()`. As stated in Section 5.3.9 residual deques remain on the `deques` tree as long as they hold the leftmost position. If `_enqueue_local_pcsws()` enqueues a deque at the leftmost position, taking the place of a residual deque, the stolen task on the residual deque is deactivated and boosted (Section 5.3.4). As explained next, this causes the stolen task to automatically relinquish the RCS and the residual deque to be dequeued from the local runqueue.

`dequeue_task_pcsws()` (Figure 5.5) implements the `dequeue_task` method of `pcsws_sched_class` and dequeues a task from any task queue[13] it might reside on.

For now, we will focus on `dequeue_local_pcsws()` (Figure 5.6), which calls `dequeue_ready_deque()` to remove a task from its current deque. Any RCS-bound task being dequeued is automatically freed of the thief RCS by way of `relinquish_rcs()`, switching to an unbound state (section 5.2.3). If deactivating a task causes a deque to become empty, then the deque itself is removed from the `deques` tree of the local `pcsws_rq` via `_dequeue_local_pcsws()`.

Apart from stealing operations[14], tasks are pushed and popped from the bottom of a deque in $O(1)$ time. Unless the deque switches from or to an empty state[15], tasks are activated and

[12] This happens when enqueueing a task causes the deque to switch from an empty to a non-empty state.
[13] A deque, the boosted runqueue, or the deboosted list.
[14] Stolen tasks are dequeued from the top of a deque in constant time.
[15] In which case the deque itself must be enqueued or dequeued from the `deques` tree.

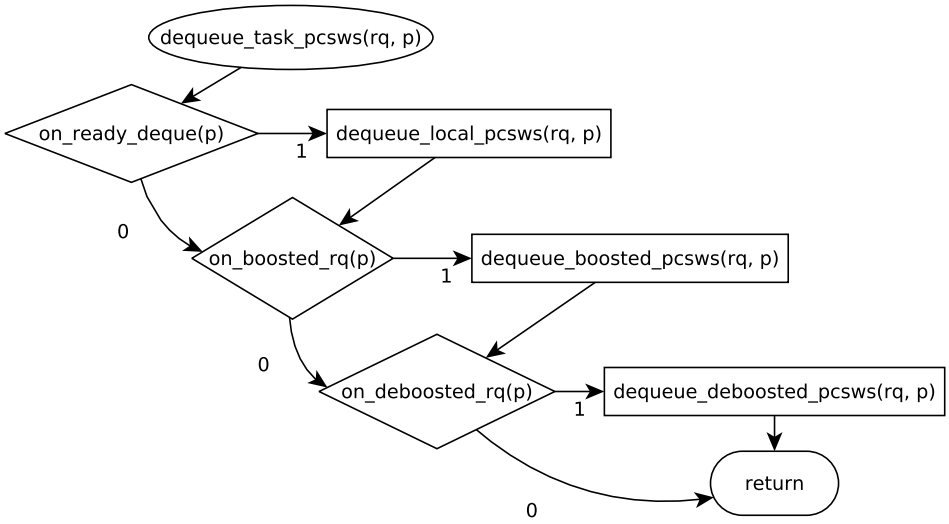

Figure 5.5: Control flow diagram for `dequeue_task_pcsws()`

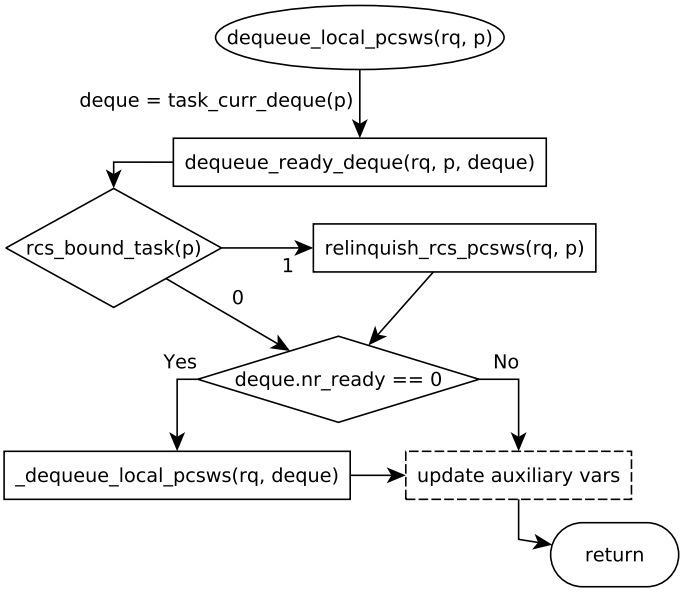

Figure 5.6: Control flow diagram for `dequeue_local_pcsws()`

deactivated at a very low computational cost. This seems particularly advantageous when tasks generate parallel regions with considerably large numbers of threads.In SCHED_DEADLINE or `fair_sched_class` for instance, activating or deactivating a task involves the manipulation of a local red-black tree subject to logarithmic $O(log(n))$ complexity, which is bound to incur substantially larger scheduling overhead.

Enqueueing or dequeueing a deque may cause changes on the auxiliary variables of `pcsws_rq` and on the `cpudl` attributes of `root_domain`. Updates on the `leftmost` and `earliest_dl` attributes of `pcsws_rq`, and on the `pcswsc_cpudl` of `root_domain`, occur whenever there is a change on the leftmost node of the red-black tree `deques`. `pushable`, `earliest_pushable_dl`, and `pcswso_cpudl`, are updated whenever an enqueue or dequeue alters the second highest pri-

ority node of the tree (the pushable deque), and unset when the tree holds less than two deques. `stealable` and `pcswss_cpudl` are updated whenever the local stealable deque changes. They are set when a dedicated deque with at least two tasks becomes the leftmost deque, or when a second task is enqueued onto the leftmost deque. The same attributes are are unset when the leftmost deque becomes a residual deque, when a dequeue on the leftmost dedicated deque leaves it with a single task, or when the local runqueue becomes empty.

5.3.3 Mapping Tasks and Deques to CPUs

Having seen how deques and runqueues are operated, we now focus on how tasks and deques are assigned to CPUs.

In Sections 5.2.3.4 and 5.2.4 we have stated that WS deques are always bound to a specific CPU through the `cpu` variable of the corresponding `ws_deque` instance (also called group CPU). The assignment of deques and tasks to CPUs is initially performed at wake up time of each task by `select_task_rq_pcsws()` (Figure 5.7).

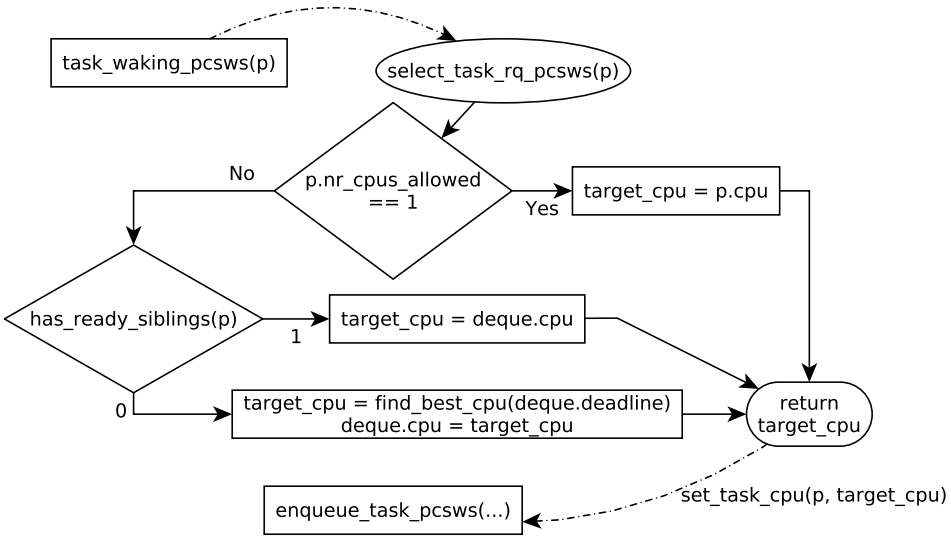

Figure 5.7: Control flow diagram for `select_task_rq_pcsws()`

For pinned tasks there is no choice but to keep them assigned to their current CPU. At enqueue time, if the task is on the wrong CPU it is boosted and returned to its group as soon as possible (Sections 5.3.2 and 5.3.4).

RCS-bound tasks do not go through `select_task_rq_pcsws()`, as they are directly assigned to the CPU of the thief RCS by `steal_task_pcsws()` (Section 5.3.9).

Other tasks are dispatched to the group CPU, as long as their dedicated deque has runnable tasks and is currently residing on a p-CSWS runqueue. If the dedicated deque is found empty at this point, *i.e.* the awakening task is currently the only runnable task to be served by the associated DS, and the dedicated deque is not on a p-CSWS runqueue, we compute a new CPU for the deque based on its priority. In the majority of cases this happens as a new p-CSWS job is released, and awakening of the master task causes the DS to switch from idle to active. Hav-

ing updated the real-time constraints for the new instance in `task_waking_pcsws()` (Section 5.3.7), `find_best_cpu()` selects the best CPU for the group according to its absolute deadline. First, `find_idle_cpu_pcsws()` tries to find an idle CPU from the `free_cpus` mask of `pcswsc_cpudl` (Sections 4.3.7.2 and 5.2.5). If all CPUs are busy, `find_latest_cpu_pcsws()` reads the binary heap `pcswsc_cpudl` and returns the lowest priority CPU, if it has a lower priority than the deque being dispatched, or the current CPU otherwise. Back in `select_task_rq_pcsws()`, we lock both the previous and next runqueues (if necessary), and set the new group CPU by assigning the returned value to the `cpu` attribute of the associated `ws_deque` instance (Section 5.2.3.4). The awakening task is later enqueued onto the deque and, as the first runnable task being enqueued, causes the deque to be enqueued onto the local p-CSWS runqueue. From that point on, each awakening task of the group going through `select_task_rq_pcsws()` is dispatched to the same CPU.

5.3.4 Boosting Tasks

Each p-CSWS runqueue defines a sub-runqueue of individual tasks, called the boosted runqueue (Section 5.2.4.2).

As explained in Section 5.3.6, boosted tasks take priority over other tasks on the runqueue, even if they have later absolute deadlines. Once a boosted task is selected for execution it runs for a single scheduler tick. Upon invocation of `task_tick_pcsws()`, the executing boosted task is removed from the boosted tree, enqueued onto the deboosted queue, and its `TIF_NEED_RESCHED` flag is set to take it off the CPU.

The *deboosted queue* is a doubly linked list, set up by the `deboosted` variable of `pcsws_boosted` (Section 5.2.4.2), containing tasks that have recently switched from the boosted state. As long as the deboosted queue is not empty, the function `__post_schedule_pcsws()` of `pcsws.c` is directly invoked by the main scheduler (Section 4.2.5.2), after the next context switch[16], to migrate all deboosted tasks to their group CPUs. If a task is still unable to migrate at this point, it is enqueued back onto the boosted tree to execute for another scheduler tick, before another migration attempt is performed. Although there is no bound upon the number of scheduler cycles that a boosted task may consume, test runs showed that boosted tasks seldom fail to migrate after a single scheduler tick.

Under the described model, although it may appear that migration of deboosted tasks implies traversal of the deboosted list, subject to linear $O(n)$ complexity, the deboosted list never holds more than one task. Anticipating future improvements, we have decided to maintain the list considering that the resulting space overhead is negligible.

5.3.5 Load-balancing

In Section 4.2.6 we have seen several examples of how load-balancing is performed in the native scheduling classes of the Linux scheduler, where two distinct paradigms stand out. The mecha-

[16] Seeing as the next context switch effectively reschedules the boosted task, the call to `__post_schedule_pcsws()` follows on the control flow of the main scheduler.

nism of `fair_sched_class` is invoked periodically to balance the workload across scheduling domains, whereas in `rt_sched_class` each CPU actively tries to exchange tasks, with other units of the root domain, at each local invocation of the core scheduler. Initially proposed by Lelli [2010], the load-balancing scheme of SCHED_DEADLINE follows the same concept, as each CPU takes the initiative to exchange tasks with other units chosen deterministically. By operating at each local scheduler tick, such an approach cannot guarantee priority correctness across CPUs at all times, it can simulate G-EDF scheduling with `HZ` resolution[17].

We have adapted the load-balancing scheme of SCHED_DEADLINE (Section 4.3.7.2) to keep the n highest priority deques (and respective tasks) allotted to the m available CPUs ($n \leq m$) as much as possible. Following the principle of SCHED_DEADLINE, in each load-balancing operation we migrate a deque that is awaiting execution on a given CPU to another runqueue where it can become the leftmost deque.

To transfer deques between runqueues two alternatives were considered:

1. Move the entire p-CSWS group at once (the deque and all contained tasks).

2. Move the deque between runqueues and migrate each task individually as it became the bottom-most task on the deque, *i.e.* the next task to be picked for execution.

Conceptually, 2 should only outperform 1 in the unlikely case when a deque be migrated back and forth between two CPU. In which case, pending tasks would still be allocated to the first CPU, and unnecessary migrations would be avoided. For the sake of simplicity we have opted to keep tasks of the same p-CSWS group together, and move the entire deque in each load-balancing operation.

5.3.5.1 Pulling a Deque

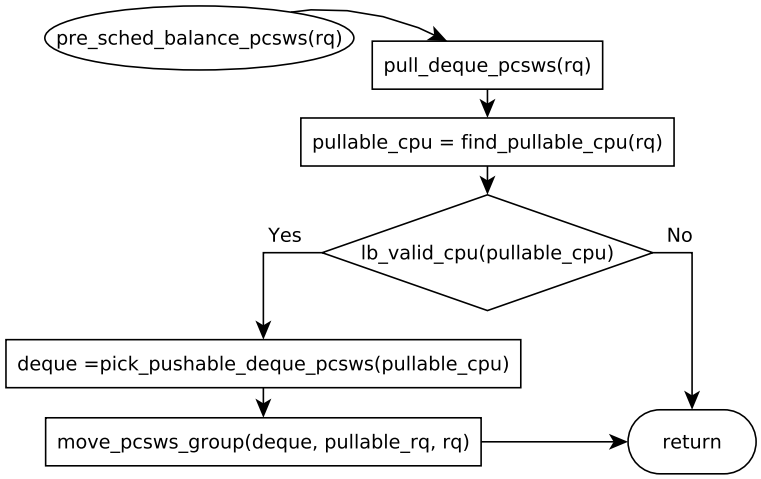

Figure 5.8: Control flow diagram for `pull_deque_pcsws()`

[17]The local CPU priority is corrected within a maximum interval of one scheduler tick, upon invocation of `pull_dl_task()`.

The first load-balancing phase is coded in `pull_deque_pcsws()` (Figure 5.8), which attempts to pull the highest priority pushable task in the system to the local CPU. Unlike `rt_sched_class`, we attempt to pull a deque to the local runqueue at each invocation of the main scheduler[18]. `pull_deque_pcsws()` is called from `pre_sched_balance_pcsws()` at the beginning of `pick_next_task_pcsws()` (Section 5.3.6), so that the bottom-most task of the pulled deque be immediately selected for execution by the current pick round.

`find_pullable_cpu()` first consults the `pcswso_cpudl` variable of `root_domain` (Section 5.2.5), via the `find_earliest_cpu_pcsws()` function of `cpudl.c`, to find the earliest deadline *pullable CPU*[19] in $O(1)$ time. Then, `pick_pushable_deque_pcsws()` follows the `pushable` pointer of the associated `pcsws_rq` to select the pullable deque.

`move_pcsws_group()` (Figure 5.9) completes the operation by moving the deque and all tasks from the pullable CPU to the local CPU. It begins by testing the affinity of all tasks in the group via `can_migrate_group()`. `_dequeue_local_pcsws()` dequeues the deque from the source runqueue, and `set_task_cpu()` is called upon each task to migrate it to the local CPU. Finally, the CPU attribute of the deque is updated, the deque is enqueued onto the local `pcsws_rq`, via `_enqueue_local_pcsws()`, and the pull operation returns.

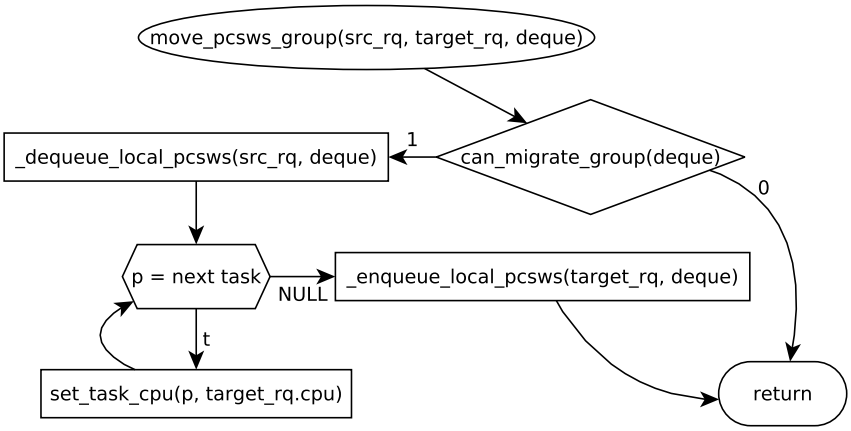

Figure 5.9: Control flow diagram for `move_pcsws_group()`

As noted in Section 4.3.7.2, SCHED_DEADLINE iterates through overloaded CPUs, identified by the `dlo_mask` bitmask of `root_domain`, until it finds a pushable task that can become leftmost on the local runqueue. We find that, although this strategy appears fast and simple, it has two significant disadvantages:

1. Task selection is not deterministic, *i.e.* the selected task is not guaranteed to have the highest priority out of all pushable tasks in the system.

2. The entire set of overloaded CPUs may be traversed before an eligible task is found. In fact, the selection mechanism cannot predict if an eligible task will ever be found.

[18] `rt_sched_class` calls `pull_rt_task()` from `pre_schedule_rt()`, which is only invoked by the main scheduler when a RT task is being deactivated.

[19] The one with the highest priority pullable task, in the system.

Considering these observations, we justify the increased contention overhead of maintaining `pcswso_cpudl` up to date, with the ability to locate the highest priority pushable deque in the system and thereby maintain an accurate Earliest Deadline First (EDF) assignment of deques to CPUs.

5.3.5.2 Pushing a Deque

`push_deque_pcsws()` (Figure 5.10) performs the inverse operation, as it tries to move the local pushable deque to another runqueue where it can become leftmost. It begins by selecting the pushable deque via `pick_pushable_deque_pcsws()`. `find_target_cpu()` then receives the absolute deadline of the pushable deque, to find a recipient CPU using the `pcswsc_cpudl` variable of `root_domain` (Section 5.2.5). In `find_target_cpu()`, if an idle CPU cannot be found via `find_idle_cpu_pcsws()`, a call to `cpudl_find()` identifies the lowest priority CPU in the system and checks if its leftmost deque has a later deadline relative to the pushable deque. `find_target_cpu()` returns the recipient CPU if a valid option is found, and `-1` otherwise.

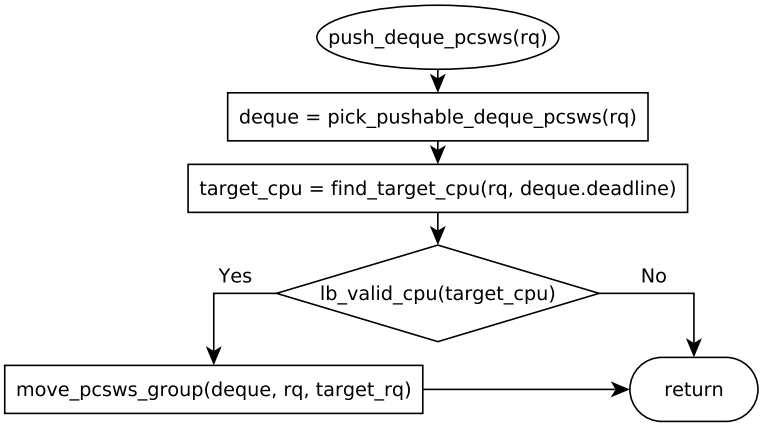

Figure 5.10: Control flow diagram for `push_deque_pcsws()`

Back in `push_deque_pcsws()`, if the returned CPU is valid, it falls upon `move_pcsws_group()` (Figure 5.9) to push the deque to the recipient CPU.

5.3.5.3 Handling Pinned Tasks

Rather than resource efficiency, interactivity, or other performance criteria, the foremost goal of our load-balancing strategy is to correct the EDF mapping of deques to CPUs.

By choosing to move the entire group in each load-balancing operation, we have stumbled upon the case of tasks that are temporarily unable to migrate. In an attempt to keep the group together we first considered aborting the load-balancing operation and trying to move the deque at a later date, but soon realized that this strategy might cause an intolerable decrease on the determinism, efficiency, and correctness of the mapping.

Tasks are automatically pinned to a specific CPU when they must respond to a given event[20] being issued on that unit, and remain as such until the event is processed to completion. Because tasks on the pushable deque are unlikely to execute soon[21], such impediment may subsist for an unpredictable, and probably large, amount of time. If we opted to keep the group together, these tasks would impede migration of the deque, and the system would incur extreme priority inversion and CPU idling.

After much deliberation, we have devised the alternative of detaching pinned tasks to allow immediate migration of the group. The affinity test in `can_migrate_group()` tests if each task of the group can migrate to the target CPU by way of `can_migrate_task_pcsws()`. If, during a load-balancing operation, one or more tasks in the group are unable to migrate between CPUs, we dislodge them from the group, boost them to ensure that they rejoin the group as soon as possible (Section 5.3.4), and proceed to move the deque and the remaining tasks[22]. In doing so, we are actually forcing priority inversion on the local CPU, but for a very short period of time rarely exceeding a scheduler tick. Although it may still have a negative impact on the real-time correctness of the system, this approach is considerably less detrimental and unpredictable in comparison with the alternative described above.

5.3.6 Selecting the Next Task

Having seen how tasks and deques are mapped to CPU, we now explain how runnable p-CSWS tasks are selected for execution, in a process that is greatly simplified by the adopted organizational design.

When the main scheduler is invoked on each CPU, it traverses the hierarchy of scheduling classes, calling the `pick_next_task` method of each class until a task is returned (Sections 4.2.4 and 4.2.5.2). This enables each scheduling class to define its own task selection policy, based on the adopted organization of tasks on their class-specific extension of the generic runqueue. In the case of `pcsws_sched_class`, this is performed by the `pick_next_task` method `pick_next_task_pcsws()` (Figure 5.11).

Before the next task is picked for execution we take the opportunity to bring the local runqueue up to date. `pre_sched_balance_pcsws()`, called at the beginning of `pick_next_task_pcsws()`, initiates the pulling mechanism described in Section 5.3.5 and the WS routine `steal_work_pcsws()` detailed in Section 5.3.9. To avoid unnecessary preemptions and migrations, pulling operations are performed preferably by idle units. After calls to `account_idling()` and `cleanup_rcs_rq()`, `pre_sched_balance_pcsws()` checks if there is pending work on the local runqueue and there exist idle CPUs in the system. If so, it skips load-balancing and WS, as idle units will attempt to pull work at their next opportunity.

In the actual selection procedure, boosted tasks have the highest priority. If the local boosted runqueue is not empty, `pick_next_boosted_pcsws()` follows the `leftmost` pointer of

[20] Examples of this include fork operations and exiting requests (when a task is set to terminate).
[21] Besides boosted tasks, only tasks leftmost on the leftmost deque are picked for execution.
[22] If the deque does not become empty as a result.

the local `pcsws_boosted` instance[23], and the leftmost boosted task is returned to the main scheduler.

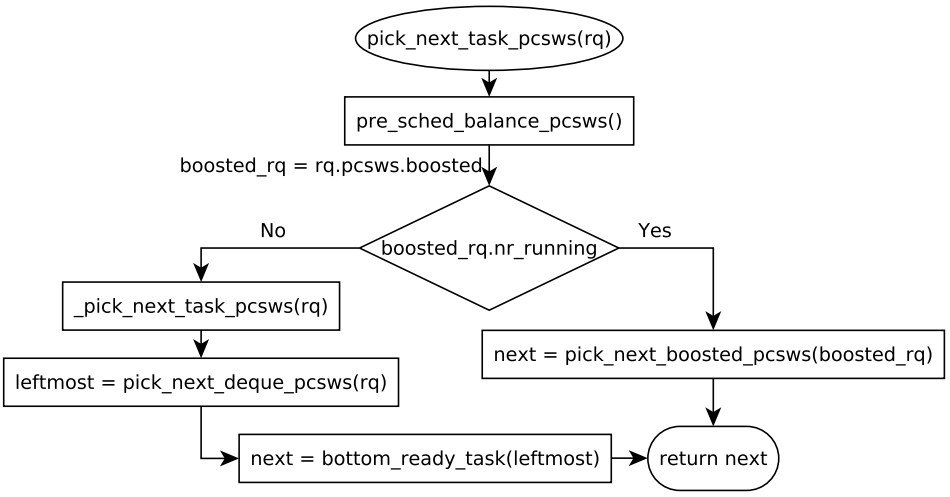

Figure 5.11: Control flow diagram for `pick_next_task_pcsws()`

If the boosted runqueue is found empty, the ordinary selection policy is put into effect. Having mapped deques to processors and organized tasks onto deques, this operation is straightforward and performed in $O(1)$ time. `_pick_next_task_pcsws()` first calls `pick_next_deque_pcsws()`, which follows the `leftmost` pointer of the local `pcsws_rq` instance to fetch the local leftmost deque. Deques on the local runqueue are guaranteed to hold at least one task, and once the leftmost deque has been identified, `bottom_ready_task()` simply returns the bottom-most task in accordance with Rule B of the p-CSWS scheduler.

If both the boosted and local runqueues are found empty, no task is returned, and the scheduler queries the next scheduling class on the hierarchy for runnable tasks.

5.3.7 Releasing the Next Job

At the end of each job, real-time tasks enter a period of inactivity until the next job is released. The `nanosleep()` system call provided by mainline Linux enables user-space processes to voluntarily delay execution until a certain instant of time passed as a parameter and expressed in nanoseconds. While this system call alone would allow users to implement the repetitive behavior of real-time tasks, we need to perform specific handling of job termination events signalized via our own system call `sched_wait_interval()`.

`sched_wait_interval()`, implemented in `kernel/sched/core.c`, takes two parameters: `*rqtp` defines the wake-up instant of the next job, whereas `*rmtp` gets the time remaining until the instant specified by `*rqtp`. We assume that is always called by master tasks at the end of each job. After managing the user-space `*rqtp` parameter through `copy_from_user()`, if the calling process is a p-CSWS task, the `wait_interval_pcsws()` function of `pcsws.c`

[23]The `boosted` attribute of the local p-CSWS runqueue (Section 5.2.4).

is invoked with the process descriptor of the calling task, as well as `*rqtp` and `*rmtp`, as parameters.

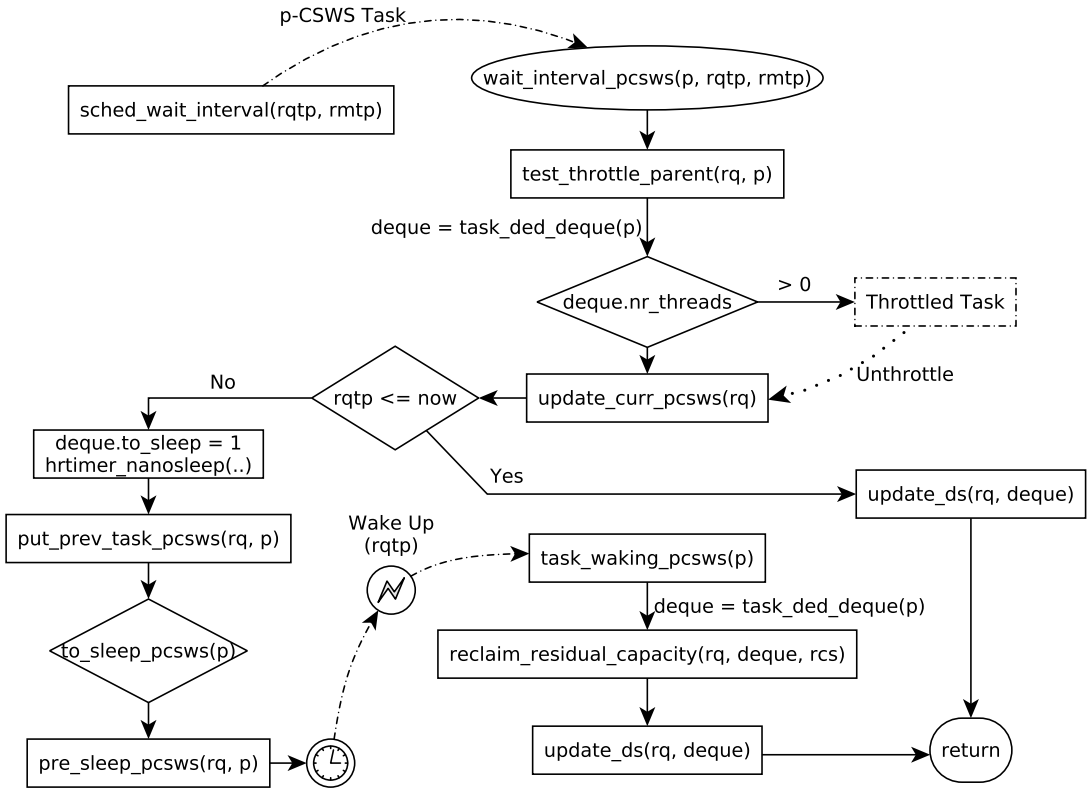

Figure 5.12: Control flow diagram for the `sched_wait_interval()` system call

`wait_interval_pcsws()` (Figure 5.12) first invokes `test_throttle_parent()` to ensure that all child threads of the calling task have completed (Section 5.3.11). After updating execution data, the flag `new_job` of the `ws_deque` owned by the task is set to indicate a new job. If the value set by `*rqtp` is in the past, this means that the next job is already delayed. In this case, according to Rule A of the p-CSWS scheduler, `update_ds()` checks if the timing parameters of the associated DS need updating, and the DS remains active. If `*rqtp` sets an instant in the future, we set the flag `to_sleep` of `ws_deque` to indicate that the current job is finishing and that the master task is close to being deactivated. A call `hrtimer_nanosleep()`, which implements the sleeping mechanism of the `nanosleep()` system call, sets the task to sleep at least until the absolute instant `*rqtp`.

To perform final response-time operations, and accurately account for the time expended between `wait_interval_pcsws()` and the deactivation instant, we hook the `__schedule()` function call that effectively takes the task off the CPU before it sleeps. In `put_prev_task_pcsws()`, `to_sleep_pcsws()` detects if the task passed as a parameter is being deactivated following a call to `wait_interval_pcsws()`[24]. If the test in `to_sleep_pcsws()` succeeds, a call to `pre_sleep_pcsws()` ensues. Here, `update_curr_pcsws()` (Section 5.3.10) accounts the

[24] `to_sleep_pcsws()` tests if the task state has been set to `TASK_INTERRUPTIBLE` and the `to_sleep()` flag of the dedicated deque is set, indicating that it has just invoked the `sched_wait_interval()` system call.

elapsed execution time, and `release_rcs()` (Section 5.3.8) releases the unused capacity on the DS as a new RCS (Section 5.3.8).

A *high resolution timer (hrtimer)* set up by `hrtimer_nanosleep` fires at release time previously specified by `*rqtp` to wake up the task. As the task is being activated, upon invocation of `task_waking_pcsws()`, we call `reclaim_residual_capacity()` to try to reclaim back the previously generated residual capacity, according to the description in Section 5.2.3.3. Finally, `update_ds()` checks if the DS should be replenished to serve the new job, in conformity with Rule A of the p-CSWS scheduler.

5.3.8 Sharing Residual Capacities

Each master task owns a `pcsws_rcs` instance established by the `residual` attribute of `sched_pcsws_entity` (Sections 5.2.3.1 and 5.2.3.3). At job completion, this instance is used to set up an active RCS with the bandwidth left unused on its sibling DS.

As explained in Section 5.3.7, RCSs are released from `pre_sleep_pcsws()` at the end of each job, by way of a call to `release_rcs()` (Figure 5.13). If the `pcsws_rcs` instance owned by the calling master task is found active, either awaiting execution or currently executing a master task, `deactivate_rcs()` deactivates it[25] and returns the leftover budget. This value is added to the budget of the dedicated deque to set the new budget for the RCS, and the absolute deadline of the dedicated deque is set as the absolute deadline of the RCS. `enqueue_pcsws_rcs()` marks the new RCS as active, by enqueueing it onto the RCS runqueue (Section 5.2.4.1). Finally, the budget of the DS is set to `0`, as stipulated by Rule D of the p-CSWS algorithm.

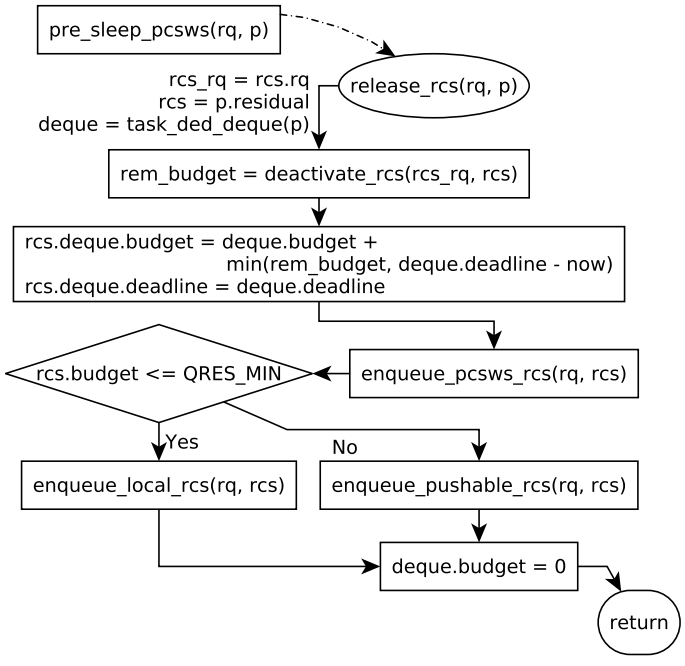

Figure 5.13: Control flow diagram for `release_rcs()`

[25] Removing it from the RCS runqueue, abandoning the stolen task, or unsetting the local `idling_rcs` pointer.

The RCS runqueue described in Section 5.2.4.1 actually consists of two distinct red-black trees of RCSs sorted by increasing order of absolute deadlines. The tree of pushable RCS `rcs_root` holds RCSs with a budget greater than the value defined by the `QRES_MIN` macro of `pcsws.c`, while RCSs with budget lower or equal than `QRES_MIN` are kept on the tree of local RCSs `local_rcs_root`. At activation time, `enqueue_pcsws_rcs()` compares the budget of the RCS with `QRES_MIN` to decide where it should be enqueued. `enqueue_local_rcs()` enqueues a RCS onto `local_rcs_root` and sets the `local` flag of `pcsws_rcs` to `1` to indicate a local RCS. `enqueue_pushable_rcs()` enqueues a RCS onto `rcs_root` and sets its `local` flag to `0`. When the leftmost node of `rcs_root`, `enqueue_pushable_rcs()` updates the `pcswsr_cpudl` instance of `root_domain` (Section 5.2.5, with the absolute deadline of the leftmost RCS.

RCSs are dequeued from the RCS runqueue by way of `dequeue_pcsws_rcs()`, which follows `dequeue_pushable_rcs()` or `dequeue_local_rcs()` depending on the `local` flag of the `pcsws_rcs` element being dequeued.

5.3.8.1 Accounting Idling Time

According to Rule I of the p-CSWS scheduler, when a local CPU becomes idle it must pull the highest priority pushable RCS and consume its budget until a p-CSWS task is assigned to the local runqueue or the residual capacity is either exhausted or expired. We accomplish this, on each CPU, through the `idling_rcs` pointer of the local `pcsws_rcs_rq`.

To implement this feature we had to go beyond natural scheduler activities and track the time expended by idle CPUs. Because the `task_tick` method is invoked exclusively for the current task, we hook the periodic scheduler with a direct call to `pcsws_tick()` from `scheduler_tick()`, regardless of the task that is executing.

`pcsws_tick()` calls `update_idling()` (Figure 5.14) to update the `idling_rcs` pointer of the local `pcsws_rcs_rq`. If `update_idling()` finds the local p-CSWS runqueue empty, it uses the `pcswsr_cpudl` instance of `root_domain` to find the earliest deadline pushable RCS in the system, and dequeues if from the `pcsws_rcs_rq` where it resides. If the local `idling_rcs` pointer is set to another RCS, this RCS is enqueued onto the local RCS runqueue[26] and `idling_rcs` is set to reference the pulled RCS. Contrariwise, if `update_idling()` finds the local runqueue busy and `idling_rcs` set, it enqueues the referenced RCS onto the local RCS runqueue, and calls `unset_idling()` to unset the `idling_rcs` pointer and cancel the timer `idling_timer`.

Once `idling_rcs` is set, `account_idling()` (Figure 5.15) is invoked at the next scheduler tick, also from `pcsws_tick()`, to decrease the budget of `idling_rcs` by the time elapsed between ticks, given by comparison of the current time with the `idling_start` variable of `pcsws_rcs_rq`. Then, if the residual capacity is exhausted or expired the RCS is deactivated via `unset_idling()`. Otherwise, the current time is stored on `idling_start` and `account_idling()` returns. Following this call to `account_idling()`, `pcsws_tick()`

[26] We ensure that the pulled RCS has an earlier deadline than the RCS currently pointed by `idling_rcs`.

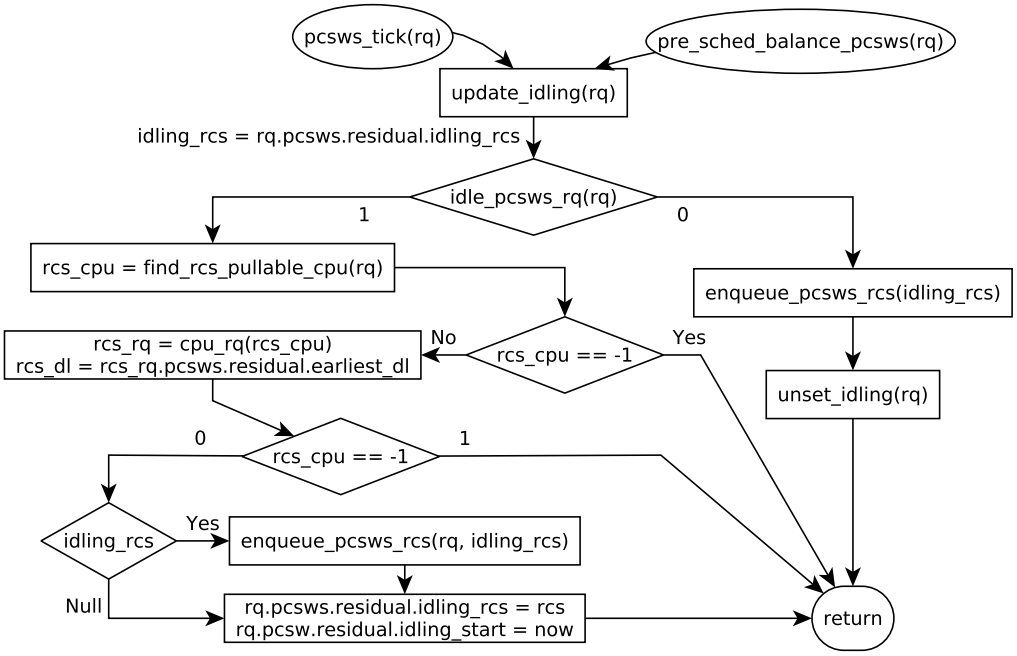

Figure 5.14: Control flow diagram for `update_idling()`

invokes `update_idling()` to check if another RCS should be pulled, as explained above.

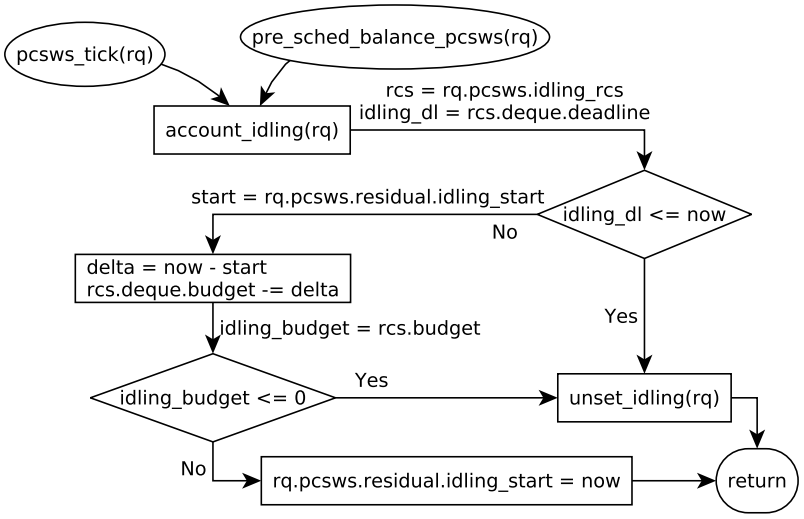

Figure 5.15: Control flow diagram for `account_idling()`

If the local p-CSWS runqueue transitions from an empty state, a call to `account_idling()` at the beginning of `pre_sched_balance_pcsws()` ensures that the elapsed idling time is accounted right before the context switch that will assign a p-CSWS task to the local CPU.

5.3.8.2 Removing Exhausted or Expired RCSs

Active RCSs must be deactivated and removed from the system once their capacity is either exhausted or expired.

As explained in Section 5.3.10, RCSs are automatically deactivated once their capacity is depleted. Expired RCSs are deactivated via `cleanup_rcs_rq()`, which simply goes through the red-black trees of RCSs removing those with a past absolute deadline. If the local p-CSWS runqueue is empty and the `idling_rcs` pointer is set, a call to `account_idling()` accounts the elapsed time and checks if the local `idling_rcs` has expired and should be deactivated. Seeing as both the local and pushable trees of RCSs are sorted by non-decreasing absolute deadlines, we keep dequeueing the leftmost node of each tree as long as its absolute deadline is in the past. As the earliest deadline RCS on the tree, if the leftmost RCS has a future absolute deadline, then all other RCSs on the tree are guaranteed to have not expired.

`cleanup_rcs_rq()` is called from `task_tick_pcsws()` at every scheduler tick. Other trigger points are `pre_sched_balance_pcsws()` (Section 5.3.6) to update the local RCS runqueue before any WS or load-balancing decisions, and `pre_sleep_pcsws()` before a new RCS is released.

5.3.9 Work-Stealing

According to Rule G once a RCS is selected for execution, it can steal the earliest deadline thread with an absolute deadline greater than that of the RCS. According to Definition 1 of the original proposal [Nogueira and Pinho, 2012], any active DS with several runnable threads and an absolute deadline greater than that of the thief RCS is considered eligible for work-stealing. In each WS operation, a thread is stolen from the earliest deadline DS from the set of eligible servers.

After careful examination we have found a significant flaw in this stategy since, by the definition, it is possible that a waiting DS be elected for WS if no busy[27] servers hold more than 1 runnable task. This would imply preemption of the lowest priority busy server to execute a thread from a lower priority DS, which is an evident case of priority inversion [Davari and Sha, 1992].

Instead, we restrict the group of DSs eligible for WS to the subset of busy DSs holding more than one task on their respective deques. Assuming that our load-balancing scheme keeps the system-wide highest priority deques executing on the available CPU, we use the heap data structure of SCHED_DEADLINE to keep a reference for the leftmost deque executing on each CPU, as long as it has several pending tasks. In every WS operation, we look at the highest priority *stealable deque* from this set, and try to steal its top-most task.

Our WS mechanism is implemented by the function `steal_work_pcsws()` (Figure 5.16) of `pcsws.c`, called from `pre_sched_balance_pcsws()` (Section 5.3.6) prior to each scheduling decision. If the local idling RCS is set (Section 5.3.8.1), the thief RCS is chosen as the one with the earliest absolute deadline between the idling RCS and the the highest priority RCS referenced by `pcswsr_cpudl` (Section 5.2.5). Else, the earliest deadline RCS of `pcswsr_cpudl` is chosen. WS operations are performed exclusively when the the thief RCS has the highest priority out of all deques on the local `deques` tree (Section 5.2.4), thus if the chosen thief RCS has a later deadline than the leftmost local deque the WS mechanism returns. Otherwise, selection of the system-wide earliest stealable deque follows, using the `pcswss_cpudl` instance of

[27] As introduced in Section 3.3.3.2, we use the term *busy server* in reference to one that is currently executing on a given CPU.

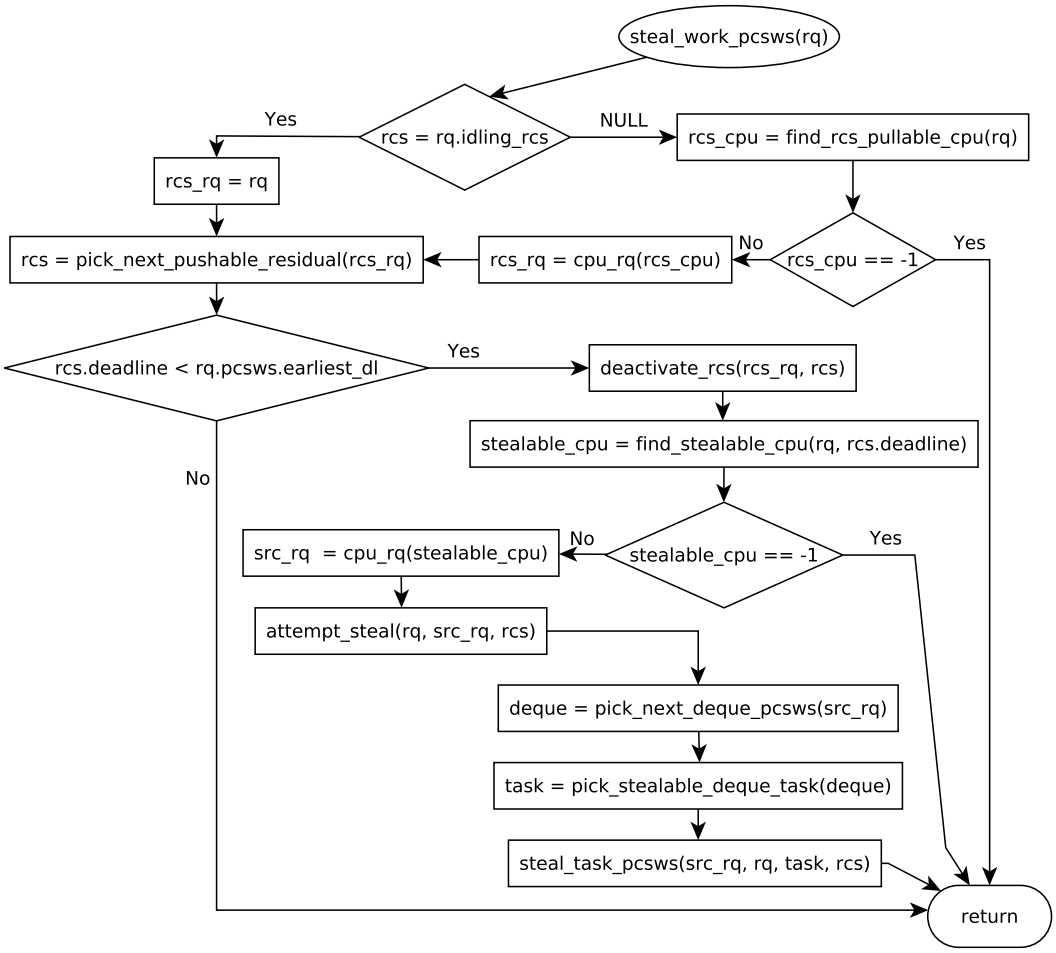

Figure 5.16: Control flow diagram for `steal_work_pcsws()`

`struct cpudl`. In `find_stealable_cpu()`, we start with the highest priority CPU and proceed to the next (in order of priority), until we find a CPU where the local `earliest_dl` value of `pcsws_rq` is greater than that of the chosen RCS. If such CPU is found, a call to `attempt_steal()` picks the top-most task of its leftmost deque, and tries to migrate it to the local runqueue. If the top-most task of the chosen stealable deque is pinned to its CPU, we boost it and iterate down the deque until an unpinned task is found. The original proposal in Nogueira and Pinho [2012] clearly states that tasks are to be stolen from the top of a deque, but we prefer to waste some processing resources trying to find a stealable task than leaving the local CPU idle or not exploiting parallelism, as failing to find a stealable task would imply. Note that choosing a stealable task from the top, or any other position in the deque, has no impact on the real-time correctness of the schedule, because all tasks in the group need to synchronize at the end of the parallel region and complete before the current job finishes. When a stealable task is found, `steal_task_pcsws()` migrates it to the local CPU and enqueues it onto the stolen list of its DS. Finally, a call to `activate_task()` enqueues the task onto the residual deque of the thief RCS, and residual deque onto the local p-CSWS runqueue (Section 5.3.2)[28].

[28] At this time, the `thief` pointer of the `pcsws_sched_entity` has already been set, causing the current deque

5.3.9.1 Returning to a DS-Bound Context

RCS-bound tasks execute as long as they are naturally deactivated, preempted, or returned back to their DS.

During the execution of a stolen task an overrun may decrease the priority of its DS (Section 5.3.10), making it ineligible for WS and forcing all of its stolen threads to yield the CPU and return to a DS-bound context. If we were to iterate through all tasks on the stolen list (Section 5.2.3.2) of a DS, migrating each task back to the local CPU, we would impose the resulting overhead upon the local CPU and delay the execution of the new leftmost deque. Also, because RCS-bound tasks are always executing on a given CPU, they must yield the CPU before migration is attempted. For these reasons, RCS-bound tasks verify if their DS remains as the highest priority active server on its local CPU at every scheduler tick. If the DS is no longer eligible for WS, the task is relinquished from its RCS, enqueued onto the local deboosted list, and migrated to the group CPU after the next context switch (Section 5.3.4).

Also, according to [Nogueira and Pinho, 2012], when a DS becomes idle and its stolen list is not empty, each stolen task is to be reclaimed back and executed until the list becomes empty. We achieve this through `reclaim_one_stolen()`, called from `put_prev_task_pcsws()` when last task of the deque is being switched off the CPU. The first task of the stolen list is fetched, enqueued onto the deboosted list of its local runqueue, and returned to the group CPU by the boosting mechanism (Section 5.3.4).

5.3.10 Accounting Execution Time

To enforce tardiness isolation among p-CSWS servers, we must track the processing time of each p-CSWS task and ensure that the reserved processing bandwidth is not exceeded.

Execution time is accounted at every scheduler tick, via invocation of `update_curr_pcsws()` from `task_tick_pcsws()`. The `se.exec_start` variable each process descriptor is traditionally used as a timestamp for the last scheduler tick. We compare the current time with this value to compute the execution time elapsed since the last tick and, after updating `exec_start` with the current time, pass the difference onto `decrease_budget_pcsws()`, where Rules B, E, and H, of the p-CSWS algorithm are implemented.

Rule E states that the capacity of a local RCS is to be donated to the next server with a later deadline executing on the same CPU. To implement this feature, we simply pick the earliest deadline local RCS and check if it has a shorter deadline than the currently executing task. If so, its budget is decreased by the execution time calculated before and passed as a parameter. If the budget is depleted as a result (`<0`), we retain the execution time in excess and jump to the beginning of `decrease_budget_pcsws()`. If the budget is not exhausted (`>0`), the function returns.

We then test if the currently executing task is unbound and executing on the wrong CPU. In this case, we add the elapsed execution time to the `overrun_amount` attribute of the associ-

of the task to be determined as the residual deque of the thief RCS. For this reason, `enqueue_task_pcsws()` automatically enqueues the task onto the residual deque (Section 5.3.2)

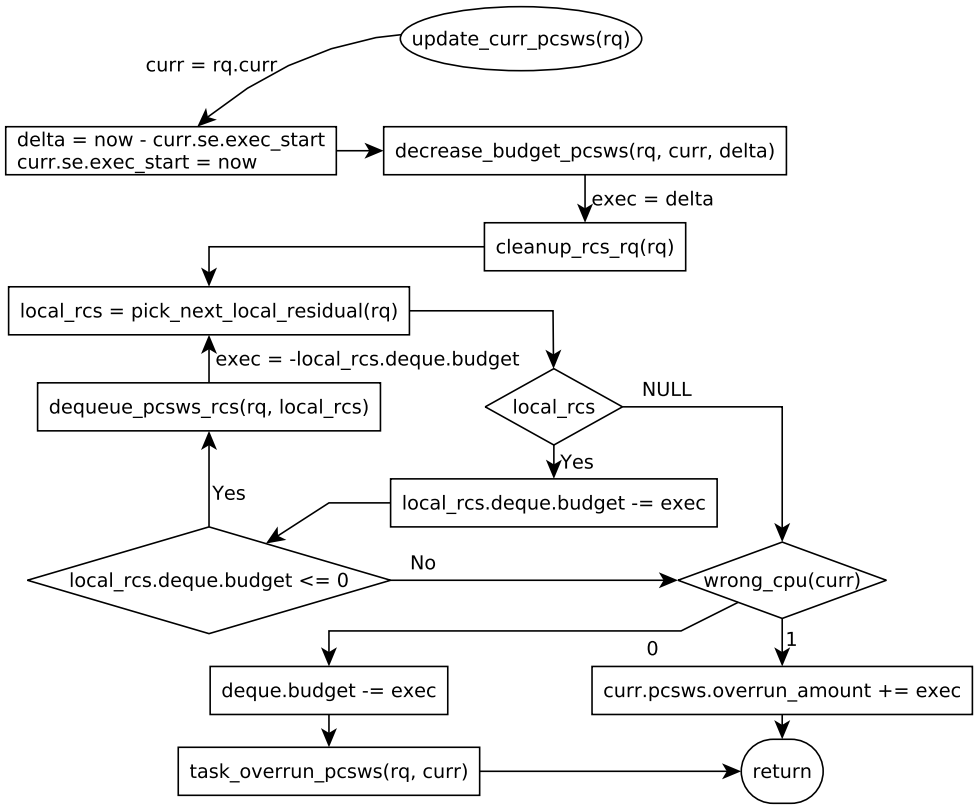

Figure 5.17: Control flow diagram for `update_curr_pcsws()`

ated `sched_pcsws_entity`, and `decrease_budget_pcsws()` returns. In Section 5.2.3.4 we have stated that all deque manipulations are protected by the runqueue lock of the currently associated CPU. If a non-stolen task is currently detached from its group, executing on another CPU, the execution time cannot be accounted directly on its dedicated deque without locking the remote runqueue. Because every unbound task is a boosted task set to return to its group very soon[29], we avoid locking multiple runqueues and account this execution time at a later instant, when the task rejoins its group. After writing `overrun_amount`, `decrease_budget_pcsws()` returns. When the boosting mechanism dispatches a task to the correct CPU, and calls `activate_task()` upon it, `enqueue_task_pcsws()` checks the value on `overrun_amount` and invokes `account_unbound_runtime()` to account the execution time overdue (Section 5.3.2). In `enqueue_task_pcsws()` the value is subtracted to the budget of the dedicated deque, and `task_overrun_pcsws()` checks if the capacity assigned to the group has been exhausted or expired, as explained in Section 5.3.10.1.

DS-bound or RCS-bound tasks, executing on the correct CPU, simply decrease the budget of their current deque conforming to Rules B and H of the theoretical proposal.

[29] Most likely after a single scheduler tick (Section 5.3.4).

5.3.10.1 Handling Overruns

If the current task is DS-bound or RCS-bound, `update_curr_pcsws()` invokes `task_overrun_pcsws()` to check if the timing constraints of its current deque have been violated, *i.e.* if the absolute deadline is in the past or the currently assigned budget is exhausted.

According to Rule B, when a DS-bound task overruns `replenish_pcsws_entity()` recharges the budget and postpones the deadline of the dedicated deque to create new server instance. Because postponing the absolute deadline of a deque may disarrange the local red-black tree of WS deques[30], `reorder_group_on_local_rq()` dequeues and enqueues[31] the deque back onto the local runqueue to rearrange the tree.

Otherwise, if a RCS-bound task executes past the residual bandwidth of is thief RCS, it is dequeued from the deque of the RCS and boosted to rejoin its group as soon as possible. The amount executed in excess is stored onto the `overrun_amount` attribute of the scheduling entity and accounted when the task returns to its group. As noted in Section 5.3.2, when a stolen task is dequeued it automatically relinquishes the RCS and switches to an unbound context.

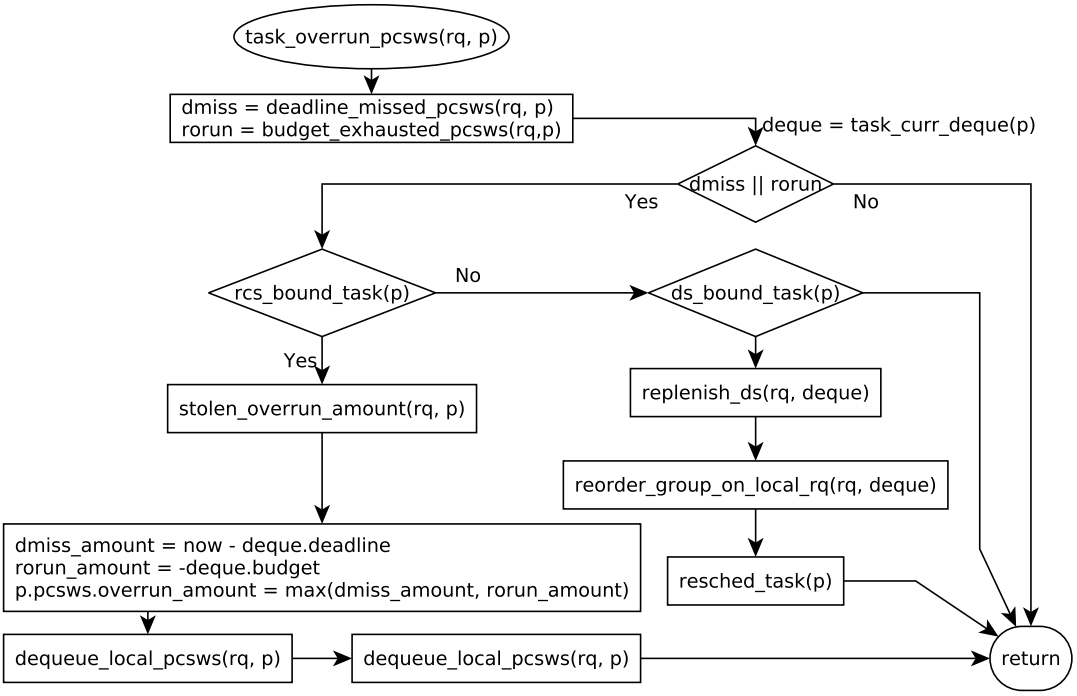

Figure 5.18: Control flow diagram for `task_overrun_pcsws()`

5.3.11 Task Termination and Parent Throttling

Under the fork/join real-time model it is expected that parallel threads finish execution before specific events (such as job completion) occur. Although the fork/join behavior is implemented in user-space, ordinary scheduler features may cause the parent task to be deactivated before all child

[30]Recall that the red-black tree `deques` of each p-CSWS runqueue `pcsws_rq` is sorted by increasing order of absolute deadlines (Section 5.2.4).

[31]Via `_dequeue_local_pcsws()` and `_enqueue_local_pcsws()` respectively (Section 5.3.2).

threads complete. To overcome this scenario and account execution time correctly, we implement a special feature, called *parent throttling*, to forcefully deactivate the master task until all child threads terminate.

The `test_throttle_parent()` function of `pcsws.c` checks the number of active threads counted by the `nr_threads` attribute of `pcsws_ds` (Section 5.2.3.2). If there exist child threads (runnable or blocked) served by the DS, we set the `throttled` flag of the master task, its `TIF_NEED_RESCHED` flag, and its state to `TASK_UNINTERRUPTIBLE`, before the main scheduler is directly invoked to deactivate the task and switch it off the CPU. Otherwise, the function returns.

`test_throttle_parent()` is called from `task_dying_pcsws()` (Section 5.2.2.1) for each master task that has just called `do_exit()`, to wait for the termination of all child threads. When `task_dead_pcsws()` (Section 5.2.2.1) is invoked for a p-CSWS thread, it decrements the `nr_threads` counter of the DS. Then, it checks if the exiting thread is the last thread bound to the DS and if the master task is throttled. If the master task has not yet been taken off the runqueue the throttling process is canceled via `unthrottle_parent()`. Otherwise, if the master task is inactive, a call to `wake_up_process()` activates the master task just as the last child thread is being switched off the CPU for the last time.

Another trigger point for `test_throttle_parent()` is `wait_interval_pcsws()`. When a master task goes to sleep after calling `hrtimer_nanosleep()`, it may release the capacity left unused on the DS as a new RCS, and force the DS budget to `0` until release time of the next job. This call ensures that the execution time of all threads is accounted before a new RCS is released. Otherwise, we could be forcing deactivation of the DS while served tasks remained active.

5.4 Summary

Chapter 6

Experimental Evaluation

"Success represents the 1% of your work which results from the 99% that is called failure."

— Soichiro Honda

6.1 Scenario

To evaluate the benefits of the proposed implementation we have initially intended to compare SCHED_PCSWS with SCHED_DEADLINE. Unfortunately, SCHED_DEADLINE is not suited for the model of parallel real-time tasks considered in this thesis (Section 5.1), as it does not support dynamic thread creation. Seeing as each SCHED_DEADLINE task runs on a dedicated Constant Bandwidth Server (CBS) reservation, adapting the implementation to our needs would either require (i) associating each dynamically generated thread to the bandwidth reservation of their parent task, (ii) running a sequential version of the user-space parallel test applications on SCHED_DEADLINE, or (iii) specifying a new bandwidth reservation for each forked thread. None of these approaches seemed viable. Option (i) implied a serious reformulation of the design and features of the implementation. Option (ii) would compare distinct user-space algorithms, and completely ignore the overheads of dynamic thread generation. Option (iii) was simply not viable, as there is no information about the execution requirements of each dynamically generated thread at runtime. Furthermore, we would need to integrate the SCHED_DEADLINE and PREEMPT_RT patches on the Linux source tree, to set it on equal grounds with SCHED_PCSWS[1].

Because we want to evaluate the benefits of parallelism, and seeing that work-first offers the best opportunities for parallelism, we have employed a work-first Work-Stealing (WS) strategy (Section 5.2.3.4) on all conducted tests.

[1] As of Linux v3.8.13, both projects were distributed as Linux source code patches, which were conflicting on multiple situations.

6.1.1 SCHED_SCBS

For these reasons, and because our base implementation is very similar to SCHED_DEADLINE, we have decided to compare SCHED_PCSWS with a simplified version of the algorithm without Capacity-Sharing (CASH) and WS features (Sections 5.3.8 and 5.3.9), henceforth referred to as SCHED_SCBS.

Each SCHED_SCBS tasks is always bound to the same dedicated server and deque, either set up by itself or by its master task. Tasks and deques are are mapped to processors exactly like in SCHED_PCSWS (Sections 5.3.2 and 5.3.3). All tasks on a deque share the same timing constraints and execute sequentially on the same Central Processing Unit (CPU), unless they are migrated by the load-balancing scheme described in Section 5.3.5. In comparison with SCHED_DEADLINE, this approach seems better suited for the model of parallel real-time tasks, since it is designed to maintain dynamically generated threads on the same CPU and explore locality of data.

To reduce unnecessary contention overhead, we have gave up caching of the stealable deque (Section 5.3.2), which involved updating the global `pcswss_cpudl` variable (Section 5.2.5).

6.1.2 Conducted Tests

The experiments expounded in the following Sections were performed on an 8-core machine of 2.0 GHz per-core, with 16 GB of main memory. At compilation of Linux v3.8.13, the following options were disabled: (i) *Group CPU Scheduler*, (ii) *CPU Frequency Scaling*, (iii) *SMT (Hyper-threading) scheduler support*, and (iv) *Tickless System (Dynamic Ticks)*. The `HZ` macro (Section 4.2.5.1) was set to 1000, and the preemption model to *Fully Preemptible Kernel*[2].

We have conducted 3 tests, each involving the execution of 20 randomly generated *server sets*. Each test was executed on 2, 4, and 8 cores, on both SCHED_PCSWS and SCHED_SCBS.

To generate bandwidth servers arbitrarily, the minimum server utilization was defined as $u_{min} = 0.1$ and the maximum utilization as $u_{max} = 0.5$. The minimum server period was set as $T_{min} = 700_{ms}$, and the maximum period as $T_{max} = 800_{ms}$. The period of each server was defined as $T_i = T_{min} + x * (T_{max} - T_{min})$, where x is a random number in the range $[0, 1]$.

A total utilization window $[U_{\prod min}, U_{\prod max}]$ was defined for each server set as follows: $[0.38, 0.40]$ for Test 1, $[0.58, 0.60]$ for Test 2, and $[0.73, 0.75]$ for Test 3[3]. The utilization of each server was dynamically computed as $u_i = u_{min} + x * (u_{max} - u_{min})$, being that $\sum_k^n u_k \in [U_{\prod min}, U_{\prod max}]$. The number of bandwidth servers n on each server set was dynamic, based on the utilization of each individual server, the defined utilization window, and the utilization bounds set by Equations 5.1.

To simulate Worst-Case Execution Time (WCET) scheduling, for each set of n servers the first $\lfloor n/2 \rfloor$ execute Hard Real-Time (HRT) tasks, each with a mean execution time of $c_i = 0.7 * C_i$, so that the real execution requirements of each job are bounded by C_i. The last $\lceil n/2 \rceil$ servers execute Soft Real-Time (SRT) tasks with a mean execution time of $c_i = C_i$. The real execution time of

[2]Provided by PREEMPT_RT.

[3]The tightness of these intervals assures a similar utilization among server sets of a given test.

each job (HRT or SRT) is randomly computed at runtime as $e_{i,j} = c_i + (0.2 * c_i - x * 0.4 * c_i)$, implying a variation of $\pm 20\%$ on the mean execution time c_i.

Each task was a simple fork/join application executing a series of NOP operations to avoid cache and memory interferences. Having simulated variable execution times for each job, although SCHED_PCSWS is designed to schedule tasks with variable parallelism we feel that a completely dynamic generation of parallel threads is not necessary to assess the performance of the implementation. The number of parallel threads n_i per-job was computed as $n_i = 1+x(m*2)$ and remained constant for each job of a Parallel Capacity Sharing by Work-Stealing (p-CSWS) task throughout execution. The real execution requirements for each thread of a job were derived as $e_{i,j,w} = \frac{e_{i,j}}{n_i+1}$. The master executed for a share $\frac{e_{i,j}}{n_i+1}$ of the total execution time $e_{i,j}$ of each job.

$U_\prod$ / m	Total servers/tasks (n)			Total threads per-job (n_i)		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	58	79	100	155	199	236
4	99	154	205	442	697	914
8	193	314	404	1774	2769	3543

Table 6.1: Composition of each test

Table 6.1 expounds the number of generated tasks and parallel threads for each test [4]. Note that as we increase the utilization window $U_\prod$, and u_{max} remains constant, the number n of generated tasks scales.

6.2 Response-Times

The main goal of this project lied with improving system throughput and boosting the efficiency of real-time applications through a heuristic exploitation of parallelism.

To evaluate the performance gains of parallel execution on each test, we have collected execution metrics on the mean response time of each task τ_i, given by $F(\tau_i) = \frac{\sum_j^{k_i} f_{i,j} - r_{i,j}}{j}$, where k_i denotes the number of jobs generated by τ_i. It is important to realize that, by measuring the time elapsed between the release time $r_{i,j}$ and completion time $f_{i,j}$ of each job, the collected data accounts not only the execution time $e_{i,j}$ of each job, but also the time spent awaiting execution, along with the overheads produced within that time frame. In order to compare the metrics experienced by SCHED_PCSWS and SCHED_SCBS, we have computed the response time ratio $F'(\tau_i) = \frac{F_{SCBS}(\tau_i)}{F_{PCSWS}(\tau_i)}$ of each task, as it was scheduled by each algorithm. The average response time ratio of each experiment was determined as $\frac{\sum_i^n F'(\tau_i)}{n}$, and compiled on Figure 6.1.

SCHED_PCSWS outperformed SCHED_SCBS on virtually every experiment [5]. Figure 6.1 shows that both algorithms perform very much alike on 2 cores, with minimal superiority for

[4] The number of parallel threads refers to the execution of a single job.

[5] Note that values over 1 indicate that, on average, the set of tasks launched by a given test responded faster on SCHED_PCSWS.

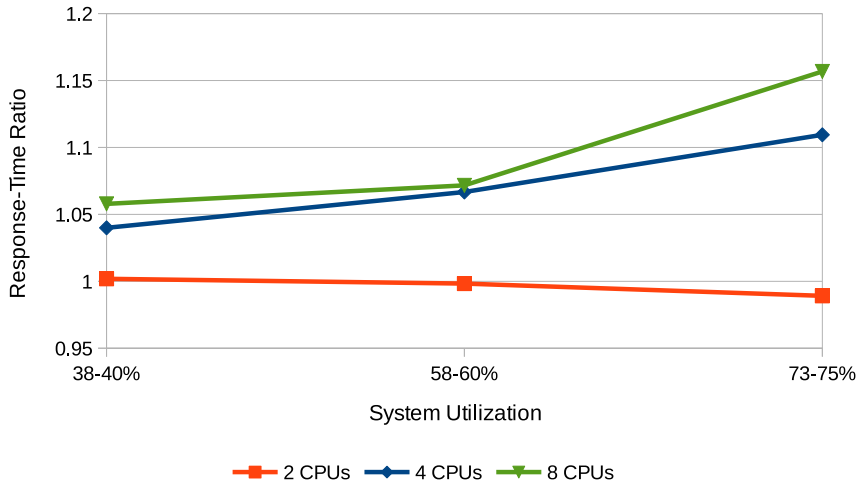

Figure 6.1: Average response-time ratio of each test

SCHED_PCSWS, except on the 73-75% utilization test where SCHED_SCBS tasks responded slightly faster. Two factors may have contributed to this outcome. First, the chances for parallel execution are likely to be scarce on 2 cores. Second, as the system gets saturated and CPUs become overloaded, stealing operations are more likely to cause preemption of a lower priority task executing on the thief CPU, improving the response-time of the stolen group, but delaying execution of the preempted task. In theory the mean execution values of each task should even out at test completion, but the depicted results are illustrative of the increased migration, preemption, and scheduling overheads, paid by SCHED_PCSWS under 73-75% utilization.

Task throughput is shown to improve steadily as the total utilization grows on 4 cores, and outperform SCHED_SCBS on every experiment regardless of the overheads generated by the WS mechanism. The benefits of parallelism are clear on the highest utilization test where, on average, tasks responded 1.1095 times faster than on SCHED_SCBS.

The results collected on 8 cores confirm the superior performance and scalability of SCHED_PCSWS, both in terms of the number of available cores, system utilization, and volume of parallel tasks and threads. Once again SCHED_PCSWS is shown to perform particularly better at the highest utilization window, as tasks responded on average 1,5677 times faster than on SCHED_SCBS.

On a final note on WCET Scheduling, although mainline Linux cannot provide predictable performance guarantees, we have confirmed that no task scheduled by the WCET missed its deadline on SCHED_PCSWS or SCHED_SCBS, attesting to the efficiency and real-time correctness of both algorithms. By these observations we can only believe that p-CSWS is a viable solution for HRT scheduling on deterministic environments.

6.3 Overheads

At each test run, we have collected statistic data on migrations and context switches, as two of the major sources of runtime overhead. First, we present the referring to the experiments conducted on 8 cores, which launched the higher numbers of parallel tasks and threads. Figure 6.2 compares

the total number of migrations triggered by SCHED_PCSWS and SCHED_SCBS.

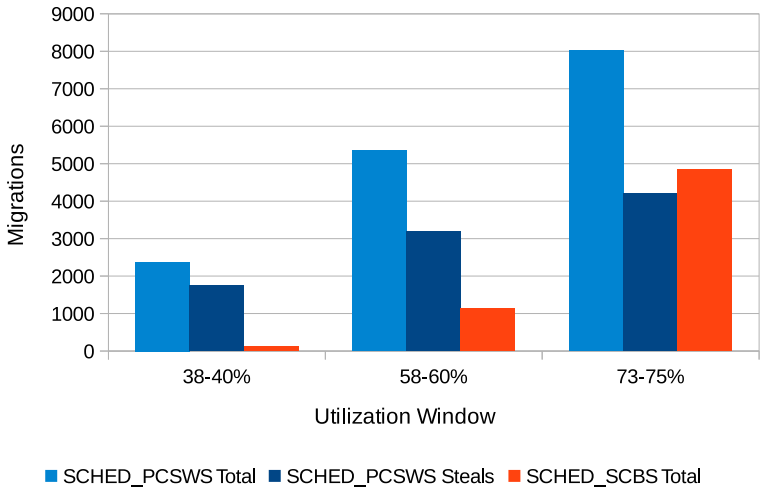

Figure 6.2: Total migrations and steals on 8 cores

Table 6.1 shows that, as we raised the utilization window, the number n of tasks increased linearly, and the volume of dynamically created threads has grown exponentially. Greater volumes of parallel tasks are bound to upset the global priority scheme more often, increasing the frequency of load-balancing operations needed to maintain priority correctness across CPUs. Indeed, the collected results show an increase on the volume of migrations as the system load grows, and a clear difference between the metrics collected for SCHED_PCSWS and SCHED_SCBS. However, the values experienced by SCHED_PCSWS appear to be tied with the observed volume of WS operations, and although migration metrics are to be taken as execution overhead, those coming from stealing operations are expected to benefit the performance of the system in two ways: (i) reduce the execution times of the stolen p-CSWS groups, and (ii) improve resource utilization.

By the linear growth on the number of stealing operations (Figure 6.2), the collected results attest the efficiency and scalability of a WS strategy, bound to generate parallelism as a function of the available CPUs, rather than the volume of active threads. As the system load increases and idle CPU become scarce, the rate of WS operations seems to decrease. By this observation we believe that the bulk of stealing operations are performed by idle cores, as planned by our implementation (Section 5.3.9).

As Figure 6.3 shows, SCHED_PCSWS and SCHED_SCBS perform similarly regarding the number of context switches involving real-time tasks. Because both algorithms ran the same numbers of tasks and parallel threads at each utilization window, we must also associate the collected statistics with the volume of WS operations raised by SCHED_PCSWS. Recall that tasks are stolen only when they are to execute at the highest priority on the target CPU, either preempting another task executing on the thief CPU, or taking execution on a previously idle unit. Interestingly, SCHED_PCSWS exhibits slightly lower context switching metrics than SCHED_SCBS under high utilization.

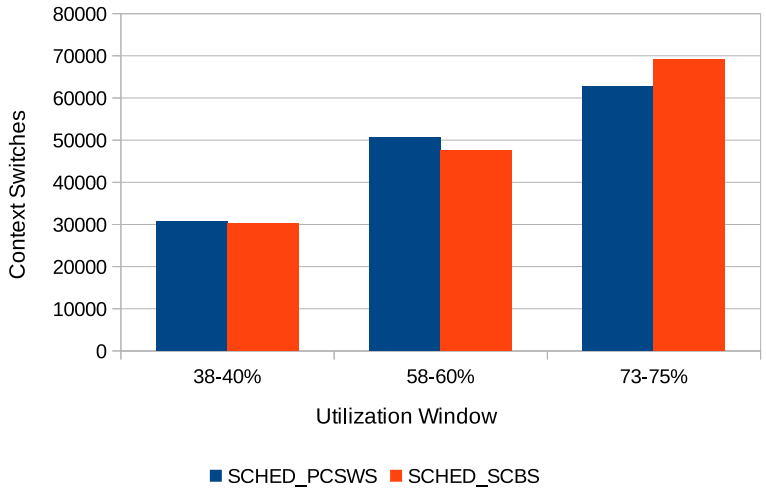

Figure 6.3: Total context switches on 8 cores

6.3.1 Scalability

To analyze the scalability of the system, we have compared the migration and context switching metrics of each test with the base readings obtained by the tests conducted on 2 cores under SCHED_SCBS. The determined scalability ratio expresses how a given metric has changed in comparison with the base reading. For instance, a scalability ratio of 2 on a 38-40% means that the metric has doubled in relation with that of the 38-40% utilization test performed on 2 cores under SCHED_SCBS.

Recall that, as shown by Table 6.1, the number of tasks grows linearly and the number of threads exponentially, as the utilization window and number of cores increase.

m	SCHED_SCBS			SCHED_PCSWS		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1.00	1.00	1.00	3.18	3.33	3.47
4	0.55	2.96	3.30	11.61	11.51	12.34
8	1.43	5.32	14.10	28.30	25.26	23.41

Table 6.2: Scale up ratios on the number of migrations

Straightforwardly, Table 6.2 shows that SCHED_PCSWS incurs a much greater volume of migrations than SCHED_SCBS, resulting from WS operations. However, these values seem to scale linearly with system utilization, while the scale up ratios of SCHED_SCBS keep increasing. Remember that WS operations always choose the highest priority eligible deque as a victim. Stolen threads immediately raise the priority of the thief CPU, by the absolute deadline of the thief p-CSWS Residual Capacity Server (RCS). Provided that, on average, each task generate enough threads to keep all CPUs busy, as parallel threads of the highest priority deques take execution on several CPU, the later deadline pushable deques are less likely to find a later deadline CPU to which they might migrate and execute straight away. In essence, this means that each WS migrations (involving only one task) contributes to reducing group migrations (involving a possi-

bly large number of tasks) particularly under high system loads. Surprisingly, SCHED_PCSWS migrations even appear to grow sub-linearly with system utilization on the 8-core tests.

Context switches occur at least twice for each generated task (or thread). Thus, like migrations, context switches are also contingent on the volume of launched tasks and threads.

m	SCHED_SCBS			SCHED_PCSWS		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1.00	1.00	1.00	1.40	1.31	1.30
4	1.02	1.15	1.31	1.26	1.51	1.45
8	2.89	1.87	1.60	2.95	2.00	1.45

Table 6.3: Scale up ratios on the number of context switches

As expected, because stealing operations always trigger a minimum of two context switches, the scale up ratios are higher for SCHED_PCSWS. However both algorithms seem to scale favorably with the increase on the number of executed tasks and threads. Referring back to Table 6.1, the volume tasks roughly doubles between tests on 2, 4, and 8 CPUs, while the number of threads increases several-fold. Regardless, context switches on SCHED_PCSWS and SCHED_SCBS grow at a less steeper rate as the number of cores is increased.

Chapter 7

Conclusion

"A problem solved, is a problem caused. That's the problem with problems, you sort one out and it makes another."

— Karl Pilkington

As real-time applications become increasingly prominent in modern day technology, under the form of Open Real-Time (ORT)

7.1 General Conclusions

The recent outbreak of multi-core chips has turned parallel processing into a main concern. In an attempt to simplify the development of parallel applications and increase productivity, parallel programming models allow developers to express parallelism by splitting their programming logic into concurrent parts, and leaving these parts to be assigned to processors by the underlying process scheduler. Under this approach known as intra-task parallelism, the scheduler must react to dynamic changes in the workload at runtime, recognizing the best opportunities for parallel execution and recomputing the mapping efficiently.

Developing efficient real-time scheduling solutions that sup- port dynamic task-level parallelism is no longer relevant only to the high performance computing niche, but is now a widespread concern, even in the embedded and real-time do- main.

Modern real-time systems generate increasingly 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 the performance of individual applications.

Furthermore, real-time systems are no longer restricted to strictly controlled dedicated environments. A new concept, formalized in the term of ORTs addresses the scheduling of real-time tasks and independently developed applications on the same shared resources, requiring additional heuristics to simultaneously guarantee the timeliness needs of real-time computations and an acceptable level of interactivity for other executions. In order to achieve these goals, the pro-

cess scheduler responsible for the assignment of processing time to tasks, must look to utilize the available resources to the fullest and extract the best performance out of the parallel platform.

Taking advantage of the modular design of the Linux scheduler, we added implemented a new scheduling policy for dynamic and irregular parallel real-time applications. Although Linux lacks in real-time predictability, we were able to prove the correctness and efficiency of our scheduler through execution data collected on extensive experimental tests. Through comparison with a sequential equivalent of our algorithm, we were able confirm the performance gains resulting from our heuristic exploitation of parallelism, and legitimate p-CSWS as a viable solution for real-time scheduling of parallel real-time tasks on open environments.

7.2 Summary of the Main Contributions

Motivated by the recent success of SCHED_DEADLINE (Section 4.3.7) and SCHED_RTWS (Section 4.3.8), this thesis undertook the scheduling of dynamic and irregular parallel real-time tasks without tardiness interferences on the Linux kernel. While SCHED_DEADLINE showed how CBS theory can be applied to the scheduling of real-time tasks without tardiness interferences on Linux, SCHED_RTWS proposed a priority-aware WS strategy to exploit intra-task parallelism on executions scheduled by the WCET.

Following the theoretical description of the p-CSWS scheduler (Section 3.3.3.2), we have combined bandwidth reservation with CASH and WS on the implementation of SCHED_PCSWS: a new multiprocessor scheduling class for the Linux kernel able to isolate the overruns of real-time tasks and scale their performance through parallel processing. To the extent of our knowledge, no research has ever tackled this issue.

Through simulation tests, SCHED_PCSWS has shown to effectively reduce the mean response times of real-time applications through parallel processing. Even though Linux is not designed as a real-time system, and unable to provide HRT determinism, none of the launched tasks missed their deadlines on the conducted test runs.

7.3 Future Work

On future developments, we intend to collect metrics on several performance hindrances of the algorithm, and study how they can be improved to benefit the efficiency of the system. Although the algorithm appeared to scale well with an increasing number of tasks and processors, believe that some of the implemented features can be refined. For instance, by migrating each task individually as a part of the load-balancing mechanism, we may be able to cut down migrations dramatically. Another point of concern lies with the 4 `cpudl` instances declared on `root_domain` and globally shared by all CPUs. By conducting the experimental tests on a maximum of 8 cores, we were not able to evaluate the impact of global contention overheads on these data structures, but we assume that they may cause degrade the performance of the system as the number of CPUs scales.

Finally, we expect to improve the achievable parallelism by resolving the implementation flaw resulting from the creation of overlapping RCSs (Section 5.2.3.3).

Bibliography

Kernel documentation on cgroups. Available at `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, a.

Kernel documentation on cpusets. Available at `https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt`, b.

Kernel documentation on scheduling domains. Available at `https://www.kernel.org/doc/Documentation/scheduler/sched-domains.txt`, c.

Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. 22:05, 2005.

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.

Adeos. Adeos.org. Available at `http://home.gna.org/adeos/`.

Gene M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *In AFIPS Conference Proceedings*, pages 483–485, 1967.

Björn Andersson and Jan Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 271, April 2002.

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.

AQuoSA Project. AQuoSA - Adaptive Quality of Service Architecture. Available at `http://aquosa.sourceforge.net/index.php`.

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.

Theodore P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical report, In International Conf. on Real-Time and Network Systems, 2005a.

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

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.

Sanjoy Baruah and Joël Goossens. Scheduling real-time tasks: Algorithms and complexity, 2003.

Sanjoy Baruah, Joël Goossens, and Giuseppe Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 154–163, September 2002.

Sanjoy K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 2004.

Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University, December 2013.

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.

Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009. ISBN 0596521537, 9780596521530.

Eric A. Brewer. Towards robust distributed systems, 2000.

Peter Brucker. *Scheduling Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2007. ISBN 9783540695158.

A. Burns. Scheduling hard real-time systems: A review, 1991.

Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009. ISBN 0321417453, 9780321417459.

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

Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of 21th IEEE RTSS*, pages 295–304, Orlando, Florida, 2000.

Marco Caccamo, Giorgio C. Buttazzo, and Deepu C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54 (2):198–213, February 2005.

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.

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.

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

F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, 2002. ISBN 9780470847664.

Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *SIGOPS Operating System Reviews*, 26:110–120, April 1992.

Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE RTSS*, page 308, Washington, DC, USA, 1997.

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.

Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. Wiley Interscience, Hoboken, New Jersey, USA, 2005. ISBN 0471467405.

D. Faggioli, M. Trimarchi, and F. Checconi. Sched_deadline. Available at `https://github.com/jlelli/sched-deadline`.

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.

João Ferreira and Luís Nogueira. Supporting server-based scheduling of parallel real-time tasks in linux. In *Proceedings of the 5th INForum*, Évora, Portugal, September 2013.

Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, 2010.

M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, c-21:948–960, 1972.

José Fonseca. Supporting intra-task parallelism in real-time multiprocessor systems. Master's thesis, Departamento de Engenharia Informática, Instituto Superior de Engenharia do Porto, September 2012.

Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201575949.

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.

M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7.

P. Gerum. The xenomai project, implementing a rtos emulation framework on gnu/linux. 2002.

H. H. Goldstein and J. von Neumann. On the principles of large scale computing machines. *John von Neumann: Collected Works*, V:1–32, 1961.

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.

R.L. Graham. Bounds on the performance of scheduling algorithms. In E.G. Coffman and J.L. Bruno, editors, *Computer and Job-Shop Scheduling Theory*, pages 165–227. Wiley, New York, 1976.

Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

Susanne E Hambrusch. Models for parallel computation. In *ICPP Workshop*, pages 92–95, 1996.

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

Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

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

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

Harry F. Jordan. Shared versus distributed memory multiprocessors. Technical report, Institute for Compter, 1991.

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.

Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010. ISBN 1593272200, 9781593272203.

Christoph Kessler and Jörg Keller. Models for parallel computing: Review and perspectives. In *PROCEEDINGS, PARS*, pages 13–29, 2007.

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

H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011. ISBN 9781441982377.

Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521876346.

Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201648652.

Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

Oh-Heum Kwon and Kyung-Yong Chwa. Scheduling parallel tasks with individual deadlines. In *Algorithms and Computations*, volume 1004 of *Lecture Notes in Computer Science*, pages 198–207. Springer Berlin / Heidelberg, 1995.

G. Kyriazis. Heterogeneous system architecture: A technical review. Technical report, AMD, 2013.

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.

Phillip A. Laplante and Seppo J. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley-IEEE Press, 4th edition, 2011. ISBN 0470768649, 9780470768648.

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

Insup Lee, Joseph Y-T. Leung, and Sang H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 1st edition, 2007. ISBN 1584886781, 9781584886785.

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

Juri Lelli. Design and development of deadline based scheduling mechanisms for multiprocessor systems. Master's thesis, Facoltá di Ingegneria, Universitá di Pisa, June 2010.

Juri Lelli, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux, 2011.

J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE RTSS*, pages 410–421, 2005.

Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th EuroMicro Conference on Real-Time Systems*, pages 193–200, Stockholm, Sweden, 2000.

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.

Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

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.

Rajib Mall. *Real-Time Systems: Theory and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2009. ISBN 8131700690, 9788131700693.

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.

P. Mantegazza, E. L. Dozio, and S. Papacharalambous. Rtai: Real time application interface. *Linux J.*, 2000(72es), April 2000.

Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 211, Toronto, Canada, 2004.

Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004. ISBN 0321228111.

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

Wen mei Hwu, Kurt Keutzer, and Timothy G. Mattson. The concurrency challenge. *IEEE Design & Test of Computers*, 25(4):312–320, 2008.

Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

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

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.

C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15:285–329, 2014.

Geoffrey Nelissen. *Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems*. PhD thesis, Université Libre de Bruxelles, 2013.

Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

Luís Nogueira and Luís Miguel Pinho. Server-based scheduling of parallel real-time tasks. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 73–82, New York, NY, USA, 2012. ACM.

OCERA Project. Open components for embedded real-time applications. Available at `http://www.ocera.org/index.html`.

Rodolfo Pellizzoni and Marco Caccamo. M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms. *Real-Time Systems*, 40:117–147, 2008.

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.

PREEMPT_RT. CONFIG_PREEMPT_RT Patch-Set. Available at `http://www.kernel.org/pub/linux/kernel/projects/rt/`.

Jelica Protic, Milo Tomasevic, and Veljko Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997. ISBN 0818677376.

M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004. ISBN 9780072822564.

Thomas Rauber and Gudula Rünger. *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010. ISBN 978-3-642-04817-3.

Mark Roth, Micah J. Best, Craig Mustard, and Alexandra Fedorova. Deconstructing the overhead in parallel applications. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization, IISWC 2012, La Jolla, CA, USA, November 4-6, 2012*, pages 59–68. IEEE Computer Society, 2012.

Politecnico di Milano RTAI, Dipartimento di Ingegneria Aerospaziale. Realtime application interface for linux. Available at `http://www.rtai.org/`.

Wind River RTLinux. Real-time linux. Available at `http://www.windriver.com/products/platforms/real-time_core/`.

Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM.

S. Sahni. Preemptive scheduling with due dates. Technical report, Department of Computer Science, University of Minnesota, 1977.

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.

Volker Seeker. Process scheduling in linux. Technical report, University of Endinburgh, 2013.

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.

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.

Sreekrishnan Venkateswaran. *Essential Linux Device Drivers*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2008. ISBN 9780132396554.

A. W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, pages 244–252, New York, NY, USA, 1987. ACM.

Xenomai. Real-time framework for linux.

Karim Yaghmour. Adaptive domain environment for operating systems. Technical report, Opersys Inc.

V. Yodaiken. The rtlinux manifesto. In *Proceeding of the Fifth Linux Expo*, Raleigh, North Carolina, Mar. 1999.