POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

An Autonomic Operating System via Applications Monitoring and Performance-Aware Scheduling

Relatore: Prof. Marco Domenico Santambrogio

Correlatore: Dott. Ing. Filippo Sironi

Tesi di Laurea Specialistica di:

Davide Basilio Bartolini

Matricola n. 745946

Anno Accademico 2010–2011

To anyone who ever taught me anything,

and to anyone I ever learnt anything from.

Contents

1 Introduction **1**

1.1 General problem Overview 1

1.2 Autonomic Computing 2

1.2.1 Self-* Properties 3

1.2.2 Autonomic Control Loop 4

1.2.3 Computing Systems Structure 7

1.2.4 Adaptivity in Operating Systems 8

1.3 Monitoring and Tracing 10

1.3.1 Time-Driver Monitoring 11

1.3.2 Event-Driven Monitoring 11

1.4 Process Scheduling 12

1.4.1 Resource Constrained Scheduling Problem 12

1.4.2 Processes, Threads, Tasks 13

1.4.3 Multitasking 14

1.4.4 Scheduling Environments and Goals 15

1.4.5 Task Classes 17

1.4.6 Scheduling Policies 18

1.5 Chapter Summary 22

2 Related Works **23**

2.1 New Design Proposals for Operating Systems 23

2.2 New Operating Systems and Autonomic Features 31

2.3 The K42 Operating System 32

2.3.1 Autonomic Capabilities Through Hot-Swapping . . . 33
2.3.2 Process Scheduling . . . 34
2.4 Angstrom Project . . . 36
2.5 Summary . . . 38

3 State of the Art in Monitoring and Scheduling 39
3.1 Runtime Monitoring Infrastructures . . . 39
3.1.1 Low Level Processor Monitors . . . 40
3.1.2 Software and Applications Monitors . . . 41
3.1.3 Monitoring in Autonomic Computing . . . 41
3.2 Process Scheduling in Linux . . . 43
3.2.1 $O(1)$ scheduler . . . 44
3.2.2 Completely Fair Scheduler . . . 45
3.3 Real Rime Schedulers . . . 46
3.3.1 Vanilla Linux . . . 47
3.3.2 Integrated Real Time and Best Effort Schedulers . . . 48
3.3.3 Real Time Operating Systems . . . 49
3.4 Adaptive Process Schedulers . . . 50
3.4.1 System Status Aware Scheduling . . . 51
3.4.2 Tasks Classification . . . 51
3.4.3 Adaptivity for Applications Performance . . . 52
3.5 Summary . . . 53

4 Proposed Approach 54
4.1 Vision and High-Level Structure . . . 54
4.1.1 Goals and Contributions . . . 55
4.1.2 Autonomic Computing System Model . . . 57
4.1.3 Autonomic Components . . . 59
4.2 AcOS: an Autonomic Operating System . . . 61
4.2.1 Monitoring Applications' Performance . . . 62
4.2.2 Heart Rate and Real Time . . . 66
4.2.3 The Heart Rate Monitor . . . 67

4.2.4 Provided Statistics 71
4.2.5 Desired Heart Rate and Performance Goal 72
4.2.6 The Performance Aware Fair Scheduler 75
4.3 Summary 78

5 Proposed Implementation 79
5.1 From Linux towards AcOS 79
5.2 Heart Rate Monitor 80
5.2.1 Overall structure 81
5.2.2 A Smart Interface for Producers and Consumers 82
5.2.3 Statistics Visualization and User Control 89
5.2.4 Structure of a Group within the Kernel 90
5.2.5 Statistics accounting 96
5.2.6 API for Producers and Consumers 99
5.3 Performance Aware Fair Scheduler 100
5.3.1 Plugging the Heuristic into the CFS 101
5.4 Summary 102

6 Experimental Results 103
6.1 Experimental Environment 103
6.1.1 Hardware Platforms 103
6.1.2 Software Test Bench 105
6.1.3 Experimental Parameters 106
6.2 Performed experiments 107
6.3 Heart Rate Monitor 107
6.3.1 Monitoring Overhead 108
6.3.2 Characterization of a Real Workload 111
6.4 Adaptive Scheduling Capabilities 112
6.4.1 Contrasting External Load 113
6.4.2 Performance Separation 115
6.4.3 Performance Inversion 118
6.5 Summary 121

7 Conclusions **122**

7.1 Concluding Remarks 123

7.2 Future works 124

List of Figures

1.1 Self-* properties taxonomy as proposed by Salehie and Tahvildari 3
1.2 Self-adaptation control loop 5
1.3 MAPE-K control loop 6
1.4 Observe Decide Act control loop 7
1.5 Main Layers in the structure of a modern computing system 8

2.1 Multikernel model structure 28
2.2 Scheduling entities in K42 35

4.1 Proposed model for an autonomic computing system 58
4.2 Autonomic components operating within the autonomic computing system model 60
4.3 Graphical representation of an hotspot 65
4.4 Design of a group of performance monitored tasks 70
4.5 Representation of the used performance goal 73

5.1 Communication protocol used for initializing and mapping the memory areas shared between kernelspace and userspace 84
5.2 Structure of an HRM group 96

6.1 Comparison between the maximum heartbeats throughput achievable with HRM with and without the cache-alignment optimization 109
6.2 Comparison between the maximum heartbeats throughput achievable with HRM and Application Heartbeats 110
6.3 Characterization of the execution of a real workload with HRM using different durations for the time window 112

6.4 Comparison of the execution traces of the global and window heart rates of a video encoder with and without high system load 114
6.5 Execution trace of the global and window heart rates of a video encoder with performance goals under high system load 115
6.6 Window and global heart rates during the execution of two applications performing the same job with no performance goals 116
6.7 Window and global heart rates during the execution of two applications performing the same job with different performance goals 117
6.8 Window and global heart rates during the execution of two video encoders running with different presets and no performance goals 118
6.9 Window and global heart rates during the execution of two video encoders running with different presets and inverse performance goals 119
6.10 Window and global heart rates during the execution of two video encoders running with different presets and inverse performance goals with a higher number of threads 120

List of Tables

5.1 HRM-related pseudo-files in the `procfs` and their use 83
5.2 Structure of a shared memory page used by HRM for counters 87
5.3 Structure of a shared memory page used by HRM for statistics and goals . . . 88
5.4 HRM-pseudo-files for displaying monitoring information and managing the groups' goal 89
5.5 *libhrm* userspace API for producers and consumers 99

6.1 System parameters used for the experiments run to obtain the presented results 106

List of Listings

5.1 Data structure representing a HRM group within the Linux kernel (defined in include/linux/hrm.h) 91

5.2 Kernelspace data structure (defined in include/linux/hrm.h) for tracking allocated shared memory pages 92

5.3 Kernelspace data structure (defined in kernel/hrm.c) for managing memory mappings 92

5.4 Declaration of the global HRM groups list (found in kernel/hrm.c) 93

5.5 Kernelspace data structures for HRM producers and consumers (declared in include/linux/hrm.h) 94

5.6 Additions to the task task_struct (in include/linux/sched.h) to support HRM producers and consumers 95

List of Abbreviations

ABI Application Binary Interface. 33

AcOS Autonomic OS. xx, 61, 62, 73, 75, 79, 81, 100, 102, 104, 105, 121, 123–126

AH Application Heartbeats. 36, 61–64, 102, 109, 110

API Application Programming Interface. 30, 33, 36, 40–43, 52, 59, 68–70, 80–83, 85, 90, 98–100

BVFT Biased Virtual Finishing Time. 49

CFS Completely Fair Scheduler. iv, 17, 21, 45, 46, 62, 75–77, 79, 101, 102, 112–114, 116, 123, 125

CHANGE Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments. 55, 56, 78, 80, 124, 125

CIL Common Intermediate Language. 31

CPO Continuous Program Optimization. 42

CPU Central Processing Unit. 1, 13–20, 29, 35, 40, 44, 45, 50, 52, 59, 64, 66, 104, 106, 123

CSAIL Computer Science and Artificial Intelligence Laboratory. 32

DRAM Dynamic Random Access Memory. 37

DVFS Dynamic Voltage and Frequency Scaling. 37

EC2 Elastic Compute Cloud. 8

ECLiPSe ECRC Common Logic Programming System. 29

EDF Earliest Deadline First. 20, 21, 50

ETH, Zurich Eidgenoessische Technische Hochschule, Zurich. 24

FCFS First Come First Served. 18, 47, 50

FIFO First In First Out. 18, 45

FOL First Order Logic. 29

fos Factored Operating System. 24–27, 29, 31, 32, 36, 37

FPGA Field-Programmable Gate Array. 27

GPGPU General Purpose computation on Graphics Processing Unit. 23

GPU Graphics Processing Unit. 27, 30

HPC High Performance Computing. 40

HRM Heart Rate Monitor. xx, 67, 68, 70–72, 74–76, 78–82, 84, 86, 87, 89, 90, 92, 94–96, 98–100, 102–113, 121–123, 125

hrtimer high-resolution timer. 97

I/O Input/Output. 17, 18, 45

IBM International Business Machines Corporation ™. 2–4, 9, 23, 32

IEEE Institute of Electrical and Electronics Engineers. 14

IPC Inter-Process Communication. 33, 63

ISA Instruction Set Architecture. 30, 31, 103

IT Information Technology. 2

LLC Last Level Cache. 104, 108

MAPE-K Monitoring, Planning, Analyzing, Executing thanks to shared Knowledge. 5–7, 10

MIT Massachusets Institute of Technology. 9, 23, 24, 32

NIC Network Interface Controller. 27

NUMA Non Uniform Memory Access. 27, 32, 37

ODA Observe Decide Act. 6, 7, 9, 10, 21, 36, 39, 55–60, 63, 66, 70, 100

OS operating system. 2, 7–10, 12, 13, 15, 16, 22–34, 39, 49, 54, 56, 57, 59–61, 122, 123

PAFS Performance Aware Fair Scheduler. xx, 75–81, 100–103, 105–107, 111–125

PEM Performance and Environment Monitoring. 42

POSIX Portable Operating System Interface for uniX. 14, 45

QoS Quality of Service. 16, 21, 52, 67, 72–77, 107, 114, 121, 123

RAD Resource Allocation/Dispatching. 48

RAM Random Access Memory. 29

RBED Rate-Based Earliest Deadline. 48, 49

RCSP Resource Constrained Scheduling Problem. 12, 13, 15

RDSL Rotating Staircase Deadline Scheduler. 45

RMS Rate Monotonic Scheduling. 20

RR Round Robin. 18, 19, 45, 47, 50

RTAI Real Time Application Interface. 49, 50

RTHAL Real Time Hardware Abstraction Layer. 50

RTLinux Real Time Linux. 49

RTOS Real Time Operating System. 49, 50

SEEC SElf-awarE Computational model. 32, 36, 37, 41, 42, 68

Sefos SElf-aware Factored Operating System. 32

SF Shortest job First. 18

SIP Software Isolated Process. 30

SKB System Knowledge Base. 29

SMART Scheduler for Multimedia And Real Time applications. 48, 49

SMT Simultaneous MultiThreading. 104

VFT Virtual Finishing Time. 45

VM Virtual Machine. 51

XML eXtensible Markup Language. 40, 42

Sommario

Gli ultimi anni hanno rappresentato un punto di svolta nelle modalità di progettazione e costruzione dei sistemi di calcolo. Le applicazioni continuano a richiedere maggior potenza di calcolo ed efficienza, ma questi obiettivi non possono essere raggiunti sultanto migliorando (ad esempio, incrementandone la frequenza operativa) una stessa architettura di sistema, come è avvenuto per lungo tempo. Anche nei sistemi embedded e portatili, l'architettura di calcolo tradizionale, basata su una singola unità computazionale centrale, sta diventando obsoleta a favore di architetture multi-core, in cui diverse unità computazionali sono a disposizione, permettendo di sfruttare l'esecuzione di task paralleli.

Questo cambiamento nella struttura delle architetture di calcolo da un processore centralizzato a un numero crescente di core indipendenti richiede un ripensamento dello stile di programmazione, per riuscire a scrivere applicazioni efficienti e affidabili in grado di sfruttare le nuove architetture. Inoltre, i requisiti in termini di prestazioni (ad esempio, a livello di qualità del servizio) e i limiti esterni (come limitazioni sul consumo di potenza o sulla massima temperatura raggiungibile) sono sempre più stringenti in un ampio ventaglio di scenari (dalle infrastrutture di cloud computing ai dispositivi mobili). Questa situazione rende difficile per gli sviluppatori applicativi, oltre che essere esperti nel loro specifico campo, rimanere anche aggiornati sulle competenze di sistema necessarie per essere in grado di ottimizzare il software per le architetture che vogliono supportare.

Una possibilità per ridurre questo sovraccarico dagli sviluppatori di applicazioni, è creare sistemi di calcolo capaci di adattare la loro struttura e il loro comportamento alle necessità delle applicazioni. Gli sviluppatori di sistema possono realizzare uno strato software di basso livello capace di dare all'architettura di calcolo capacità di auto-adattamento, permettendo ai sistemi di calcolo di perseguire sia l'obiettivo di raggiungere massime prestazioni sotto diverse

condizioni operative sia di mantenere il proprio stato all'interno di un confine desiderato (per esempio, in termini di consumo di potenza, temperature, ...). Un sistema capace di esibire questo comportamento è detto autonomico, o auto-adattativo e ricercatori con background in informatica e teoria del controllo si sono aggregati in una comunità di ricerca che si occupa di creare innovazione in termini di capacità dei sistemi di calcolo di auto-gestire i propri parametri di esecuzione.

La struttura di quasi quasi tutti i moderni sistemi di calcolo può essere suddivisa in tre livelli principali: i componenti hardware, il sistema operativo e le applicazioni. Il livello dell'hardware è piuttosto eterogeneo e i suoi componenti sono raggiungibili attraverso interfacce di basso livello che sarebbe troppo complesso utilizzare direttamente. Il sistema operativo si occupa di gestire l'hardware ed esporre delle interfacce di più alto livello alle applicazioni. Le applicazioni utilizzano le astrazioni offerte dal sistema operativo per raggiungere gli obiettivi per cui il sistema di calcolo viene utilizzato. Poiché il sistema operativo è il componente che è incaricato di gestire le risorse di sistema, esso è il primo strato su cui si deve lavorare per permettere al sistema di ragggiungere caratteristiche autonomiche, come descritto sopra. Per estendere il concetto di sistema operativo in questa direzione, è necessario adottare un sistema di controllo basato su un anello in retroazione, all'interno del quale dei monitor raccolgano informazioni riguardo lo stato del sistema e il suo ambiente e dei motori decisionali siano in grado di analizzare questa conoscenza per mettere in atto azioni correttive per mantenere il sistema all'interno dello spazio di stato desiderato.

Uno dei componenti del sistema operativo che ha maggior impatto sul comportamento a runtime di un sistema di calcolo è lo scheduler dei processi. Di fatto, lo scheduler è il componente che determina in che modo le capacità computazionali offerte dall'architettura di calcolo vengano assegnate alle applicazioni in esecuzione. Questo compito è diventato di grande importanza con il supporto del multitasking nei vecchi sistemi con singolo processore ed è diventato ancora più cruciale con l'introduzione di processori multicore e capaci di supportare più thread in parallelo (SMT).

L'idea che ha dato origine a questa tesi è la creazione di uno scheduler capace di valutare le prestazioni delle applicazioni e di adattare le proprie decisioni basandosi sulle necessità, in termini di prestazioni desiderate, delle applicazioni stesse. Per realizzare questa idea, il primo passo è la creazione di un monitor per raccogliere le informazioni riguardo le prestazioni

delle applicazioni e la definizione di una modalità per definire degli obiettivi prestazionali. In seguito, va determinata una politica di adattamento capace di confrontare le prestazioni misurate con quelle desiderate e di agire sullo scheduler modificandone le decisioni per permettere alle applicazioni di raggiungere gli obiettivi predefiniti.

Lo scopo del lavoro prosentato in questa tesi è quello di sviluppare tutti questi passi, progettando una metodologia per dotare un sistema operativo di capacità di auto-adattamento e implementando uno scheduler adattativo capace di prendere in considerazione lo stato del sistema e gli obiettivi prestazionali della applicazioni nel decidere come assegnare alle applicazioni in esecuzione le risorse computazionali a disposizione. Più nel dettaglio, il primo contributo di questa tesi è la formalizzazione, attraverso la definizione di termini appropriati e di una metodologia, di un approccio di alto livello al problema di creare un sistema operativo autonomico migliorando quelli esistenti. Il lavoro proposto in questa tesi, però, non si limita a formalizzare e presentare la metodologia e va oltre, progettando un sistema operativo autonomico (denominato AcOS) e implementandolo come una estensione del kernel Linux. La fase implementativa è focalizzata sulla creazione dei primi due componenti autonomici di AcOS: una interfaccia di monitoring delle applicazioni, chiamata "Heart Rate Monitor" e una estensione dello scheduler, chiamata "Performance-Aware Fair Scheduler", che utilizza il monitor per modificare il meccanismo di scheduling a seconda delle prestazioni e degli obiettivi delle applicazioni. I risultati di questa implementazione dimostrano che l'approccio propost è realizzabile e pongono le basi per lavori futuri, che si focalizzeranno sull'estensione di AcOS con altre capacità autonomiche.

Questa tesi descrive il lavoro brevemente illustrato in questo Sommario, dallo studio iniziale dello stato dell'arte fino all'implementazione, test e valutazione del sistema proposto. La trattazione è suddivisa nei seguenti Capitoli (in lingua inglese):

- Il Capitolo 1 illustra i concetti principali utili a meglio comprendere il resto del documento.
- Il Capitolo 2 presenta alcuni lavori correlati nell'area dei sistemi operativi, focalizzando l'attenzione su quelli legati a caratteristiche autonomiche.
- Il Capitolo 3 dà una visione dello stato dell'arte riguardo a monitoring e scheduling che è stato considerato per progettare il sistema proposto.

- Il Capitolo 4 offre una presentazione dei contributi di questa tesi in termini di definizione di una metodologia e progettazione del sistema di monitoring e dello scheduler.
- L'implementazione che è stata realizzata su Linux è dettagliatamente descritta nel Capitolo 5 e il Capitolo 6 propone alcuni risultati sperimentali che validano e caratterizzano il laovoro svolto.
- Infine, il Capitolo 7 offre alcune osservazioni conclusive e suggerisce possibili direzioni per i futuri lavori che saranno sviluppati su AcOS.

Summary

The last decade signed a turning point in the way computing systems are engineered and built. The applications still require ever-increasing computing power and efficiency, but this cannot be achieved by just enhancing (e.g., by increasing the operating frequency) a well-established architecture, as it happened before. Even in embedded and mobile devices, the traditional architecture based on a single central processing unit is becoming obsolete in favor of multi-core architectures, where different processing cores are available for leveraging parallel computation.

This change in the structure of the computing architectures from a centralized processing unit to several independent cores requires a reworking of the programming style needed to produce efficient and reliable applications for supporting the new architectures [35]. Moreover, performance requirements (e.g., in terms of quality of service) and runtime bounds (e.g., limits on power consumption or working temperature) are more and more specific and demanding in a variety of scenarios (from cloud computing facilities to embedded and mobile devices) [40]. This situation makes it hard for application developers to retain, beyond the expertise in the specific domain of the application, also competence in tuning their software for the underlying architecture.

A possibility for relieving this overburden from application developers is to devise computing systems equipped with the capability of adapting their structure and behavior to the needs of the running applications. System developers can create a low-level layer able to enhance the bare computing architecture with self-adaptive capabilities, enabling the computing system to both pursue the goal of maximum performance under any working conditions and maintain its status inside any wanted boundary (e.g. power consumption, system temperature, . . .). A system capable of this behavior is named *autonomic*, or *self-adaptive* and researchers from

computer science and control have gathered in an autonomic computing research community able to build innovation in the self-management of computing systems [41].

The structure of almost all the modern computing systems can be divided in three main layers: the hardware components, the operating system and the applications. The hardware layer is quite heterogeneous and its components export very low-level interfaces that would be too complex to be directly used by the applications. The operating system takes care of managing the hardware layer and to export more suitable interfaces to the applications. The applications use the system resources exposed by the operating system to execute the tasks the system was built for. Since the operating system is the component that is in charge of managing the resources of the computing system, this is the first system layer that should be extended to enable the system towards an autonomic operation, as described above. To extend the concept of operating system in this direction, the current *open loop* operation mode must be changed to a *closed loop* control. In such control loop, *monitors* allow the system to gather knowledge about its status, a *decision* engine analyzes these data and evaluates possible actions that are provided by *actuators* that can alter the system status according to its needs.

One of the operating system components which has a great impact on the runtime behavior of the overall system is the process scheduler. In fact, this is the component that decides how the processing capabilities offered by the underlying architecture are to be allotted to the running applications. This activity became fundamental when multitasking uniprocessor systems were introduced and has become even more important with the introduction of multicore and simultaneous multithreading enabled processors.

The idea that gave birth to this thesis is to create a performance aware adaptive process scheduler able to take into account the current and desired performance of the running applications when deciding how to assign the computing resources. In order to realize this idea, the first step is to provide the operating system with a *monitor* to gather information on the current performance of the running applications and a means of defining performance goals for these applications. Then, it is necessary to determine a suitable way for acting on the scheduler and an *adaptation policy* able to take decisions about what actions are to be taken based on the current measured performance and the performance goals.

The aim of the work presented in this thesis is to develop all of these steps, designing a

methodology for enhancing a commodity operating system with self-adaptive capabilities and implementing a performance aware adaptive scheduler capable of taking into consideration the status of the system and the performance goals when deciding how to share the available computing resources among the running applications. More in details, the first contribution of this thesis is the formalization, by defining a suitable vocabulary and a methodology, of a high-level approach to the problem of enhancing a commodity operating system with autonomic capabilities. The work proposed in this thesis, however, goes beyond the mere formalization of a methodology by actually designing an autonomic operating system (named AcOS) and implementing it as an extension of the Linux kernel. The implementation is focused on the creation of the first two autonomic components in AcOS: a general-purpose software monitoring system, called “Heart Rate Monitor”, and an extension to the process scheduler, named “Performance Aware Fair Scheduler”, which uses the monitor to bring performance-awareness into the scheduling decisions. The results of this implementation demonstrate that this approach is applicable and lay a base for future works focused on extending AcOS with more autonomic features.

This thesis describes the work briefly outlined in this summary, from the initial study of the state of the art to the implementation, test and validation of the proposed system. The discussion is divided into the following Chapters:

- Chapter 1 is meant for illustrating and defining the key concepts needed to better understand the remaining of the document.
- Chapter 2 presents an overview of some interesting related works in the area of operating systems, with focus on those featuring autonomic capabilities.
- Chapter 3 outlines the state of the art in monitoring and scheduling analyzed in a preliminary study for designing the proposed monitor and scheduler.
- Chapter 4 contains a presentation of the contributions of this thesis in terms of definition of a methodology and design of the proposed monitor and scheduler.
- Chapter 5 gives a thorough view of the work done for implementing the proposed system over Linux and experimental results validating this system are presented in Chapter 6.
- Finally, Chapter 7 comes to some conclusive remarks and suggests possible directions for future works on AcOS.

Chapter 1

Introduction

The work proposed in this thesis is related with different research areas (operating systems, monitoring, process scheduling), with the ideas of autonomic computing keeping the different parts together. An introduction to some concepts in these areas is provided in this Chapter. First, an overview of the general problem of exploiting the new computing architectures subject to diverse requirements is proposed in Section 1.1, giving an initial motivation for this thesis. Then, some definitions in the area of adaptive systems, which may be used to tackle this problem, are recalled in Section 1.2, focusing on the area of operating systems. Finally, Section 1.3 gives some relevant concepts about monitoring and tracing techniques and Section 1.4 contains a brief dissertation about the scheduling problem, focusing on the role of the *process scheduler* in operating systems.

1.1 General problem Overview

The evolution of information management in human activities has led to the contemporary *information society*, where computing systems and devices gained a pervasive presence in all the aspects of modern life. This trend, together with the ever increasing performance demands from the applications, has determined, in the last decade, a turning point in the structure of the computing architectures. The traditional structure with a single Central Processing Unit (CPU) in charge of performing all the computations has been pushed to its limits and, in order to still improve the performance, multicore architectures have become the new mainline paradigm. The evolution of computing systems has led from isolated single

machines that took a whole room to an intricate worldwide network made of connections between heterogeneous devices (from embedded systems and handhelds to supercomputers, through personal computers and laptops) constantly connected and communicating [40] and the multicore paradigm is rapidly taking over in the architectures of all these devices, even in an embedded and mobile context. This change towards a parallel architectural paradigm makes it more difficult for application developers to produce efficient and portable software and introduces difficulties at the operating system (OS) level in enabling the applications to benefit from all the capabilities of these new architectures [76].

The increased complexity of the computing systems, from the architectural to the infrastructural level, is leading to a scenario where computing systems will be beyond human ability to efficiently managing them [40]. This trend of increasing complexity is very difficult to contrast just with the usual approaches of (software) engineering or by seeking innovations in programming methods; a brand new approach is needed to keep the Information Technology (IT) infrastructures manageable and make them more efficient in serving our automation needs.

1.2 Autonomic Computing

A promising approach to address the problem exposed in Section 1.1 is to move the burden of managing the computing systems complexity into the computing systems themselves, making them able to self-manage their resources by autonomously making low-level decisions according to high-level goals specified by the systems' users. Such approach has been proposed by Paul Horn (from International Business Machines Corporation ™ (IBM)) under the name of *autonomic computing*. In 2001, Horn published a manifesto [40] that outlines the target characteristics of computing systems built following the autonomic paradigm. The term '*autonomic*' was chosen referring to the autonomic nervous system in biological life, which is in charge of controlling the unconscious actions in a living body. For instance, in humans, the autonomic nervous system monitors and controls the heart rate, the body temperature and a number of other vital parameters to ensure to the body a steady internal state called '*homeostasis*'. What is really interesting about the autonomic nervous system, is that it does all of its work '*in background*', without any of us being aware of its activity. This permits us to

focus on our daily tasks, without having to care about all these low-level though fundamental functions. This is how autonomic computing systems are intended to behave: they must be able to internally manage their resources, exposing a simple high-level interface for the users to express the goals they want the system to pursue.

1.2.1 Self-* Properties

For a system to behave according to the autonomic paradigm, it needs to present some properties, which basically involve some sort of knowledge of its internal status and of its working environment. These properties are needed for enabling the system to be *self-adaptive* [62] (i.e. able to adjust itself to changes happening during operation), which is the base for presenting an autonomic behavior, and are referred to as '*self-* properties*'. Some of these self-* properties are already mentioned in the IBM manifesto [40] and are further formalized by Kephart and Chess [47]; a more complete hierarchical taxonomy is provided by Salehie and Tahvildari [62], who identifies three levels of self-* properties under the domain of self-adaptive software. The hierarchical view proposed by Salehie and Tahvildari is valid beyond the boundaries of self-adaptive software and can be applied to generic autonomic systems; this taxonomy [62] is proposed represented in Figure 1.1. The Figure lays down the different self-* properties in

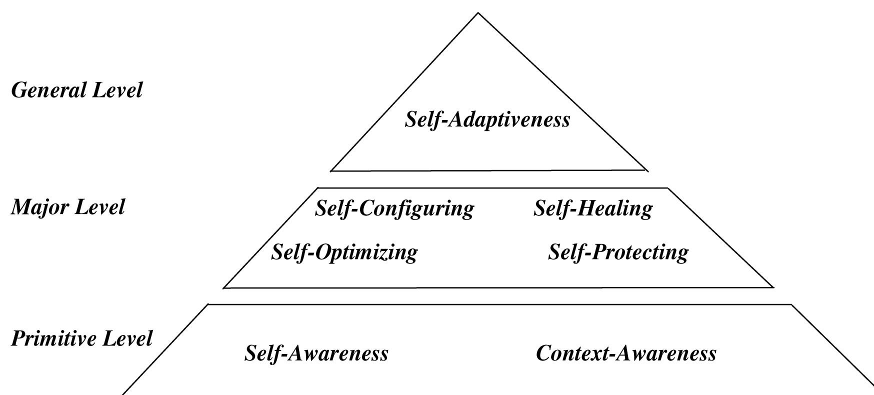


Figure 1.1: Self-* properties taxonomy as proposed by Salehie and Tahvildari [62]

a pyramid, showing a hierarchical organization divided into three levels:

- A "*general level*" of self-* properties includes *self-adaptiveness* and *self-organization*, where the former property refers to the system as a whole entity able to modify its

behavior according to the working conditions and the latter emphasizes the system being formed by (semi-)independent modules able to orchestrate their work to achieve a given goal.

- A "*major level*" includes the properties initially identified by the IBM autonomic computing initiative, which go under the category of *self-management*:
  - *Self-configuration* is the ability of the system to automatically configure itself according to high-level policies.
  - *Self-optimization* enables the autonomic system to autonomously tune its working parameters to always yield the best performance.
  - *Self-healing* is the capability of detecting, diagnosing and repairing localized problems in both software and hardware.
  - *Self-protection* targets the ability of the autonomic system to defend the system against incoming attacks and to anticipate problems, acting to mitigate or completely avoid them.
- A "*primitive level*" defines the basic properties needed in order to support the ones classified in the major and general levels:
  - *Self-awareness* means that the system possesses knowledge of its internal state and behavior, which is gathered through self-monitoring.
  - *Context-awareness* indicates knowledge by the system of the current conditions of the context it is operating in.

This taxonomy identifies a large set of self-* properties that are desirable in a generic autonomic system. Clearly, not all the self-adaptive systems need or can implement all of these properties; for instance, self-awareness may be enough to allow self-adaptiveness and self-optimization in a system put in a protected environment, while context-awareness may be more desirable for a system working in an unpredictable environment.

1.2.2 Autonomic Control Loop

In order to build a system able to expose the self-* properties outlined in Section 1.2.1, a new paradigm of how a system is to be controlled must be introduced. This paradigm

is characterized by a recurrent sequence of actions which consist in gathering information about itself and its surroundings (gaining self- and context-knowledge), evaluating the new information and acting in order to react to any relevant change. This operational scheme constitutes a control loop typical of autonomic systems. In literature, there exist various definitions of this control loop (at least three, which are shown here), each of which highlights some peculiarities of how the loop works. A first version of the autonomic control scheme is named *Self-adaptation control loop [62]* and it is represented in Figure 1.2. This representation

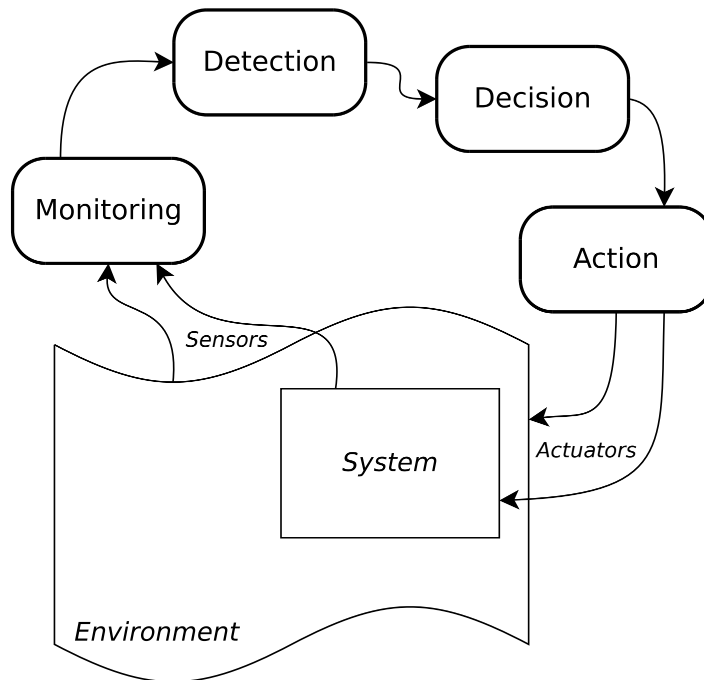


Figure 1.2: Self-adaptation control loop

emphasizes the separation between the detection and decision phases. The detection process is in charge of analyzing the data coming from the sensors and to detect when something should be changed in order to restore the system from an unwanted state into its desired working conditions. The decision process is in charge of determining what should be changed, i.e., picking the right action to be performed. A second version of the autonomic control loop is called *Monitoring, Planning, Analyzing, Executing thanks to shared Knowledge (MAPE-K)* [41, 47] and it is represented in Figure 1.3. When an autonomic element is described by means of the MAPE-K representation, the component which implements the control loop is

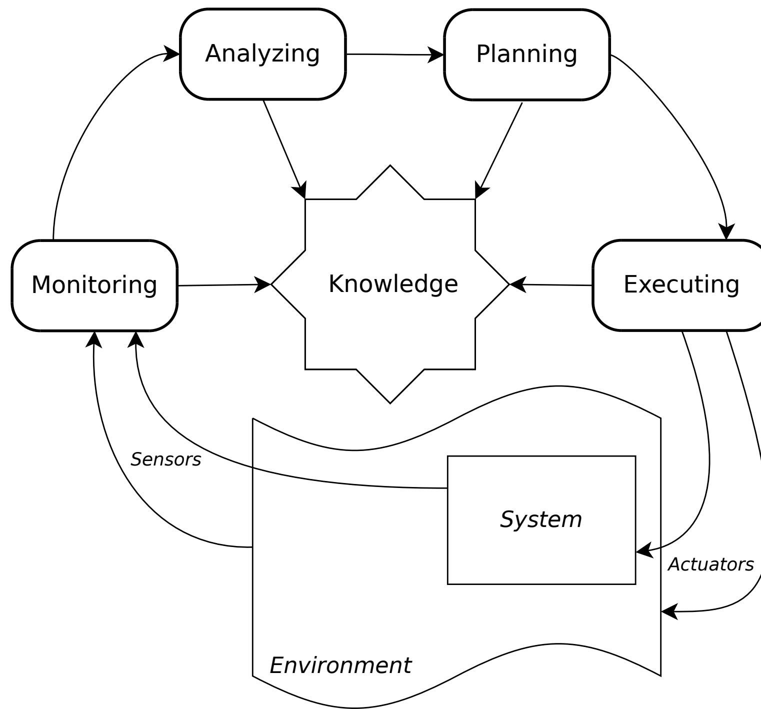


Figure 1.3: MAPE-K control loop

referred to as the *autonomic manager*, which interacts with the *managed element* by gathering data through sensors and acting through actuators (or effectors) [41]. This control scheme emphasizes the fact that a shared knowledge about the system and its environment must be detained in order to successfully execute the autonomic control scheme. A third version of the autonomic control loop is named *Observe Decide Act (ODA) loop* [66] and it is represented in Figure 1.4. This representation is more general with respect to the MAPE-K and Self-adaptive schemes and, being more generic, it summarizes the essence of the autonomic control loop. The steps of the ODA loop are *observation* of the internal and environmental status, *decision* of what action (or whether no action at all) is to be taken, based on the observations and *action*, i.e., perturbation of the internal or external status in order to modify it towards a better condition for the system. More in details, the stages of the ODA loop are characterized as follows:

- The observation phase is generally accomplished by using *monitors* (for instance, thermometers, throughput meters, latency measures, event counters . . .) to gather information about the environment and the internals of the system.

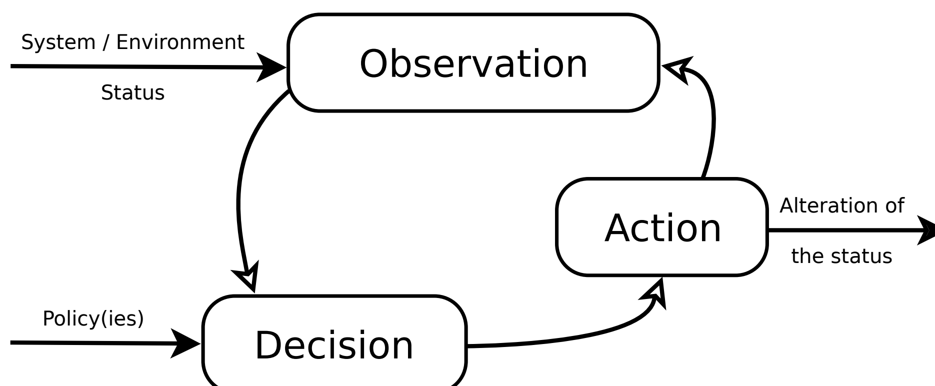


Figure 1.4: Observe Decide Act control loop

- The decision phase takes into account the data gathered through the observation and an additional input representing the decision policy(ies) implemented in the system, which can be based on different techniques including control theoretic controllers, neural networks, machine learning agents, ...
- The action phase is performed through *actuators*, which are devices (virtual or physical) which allow the system to alter its internal status or the operating environment.

The ODA loop is the minimal representation for the class of control loops (including the MAPE-K and Self-adaptive schemes) that can be used to equip a system with self-adaptive properties; thus, in this document, this loop will be kept as the reference for how autonomic systems are controlled. In an autonomic system, the ODA control loop may appear at different levels, where each component of the system (e.g., hardware modules in a computing system) is thusly controlled at a lower level and there is a higher-level controller (e.g., a software coordinator) which orchestrates the modules towards the specified goals.

1.2.3 Computing Systems Structure

Most of the modern computing systems can be modeled according to a common general structure consisting of three layers: *Hardware Components*, *operating system* and *Applications*; this structure is represented in Figure 1.5. This representation is very general and different computing systems present variations of this structure where the thickness of each layer depends on the characteristics the system is designed for. For instance, an embedded device targeted for

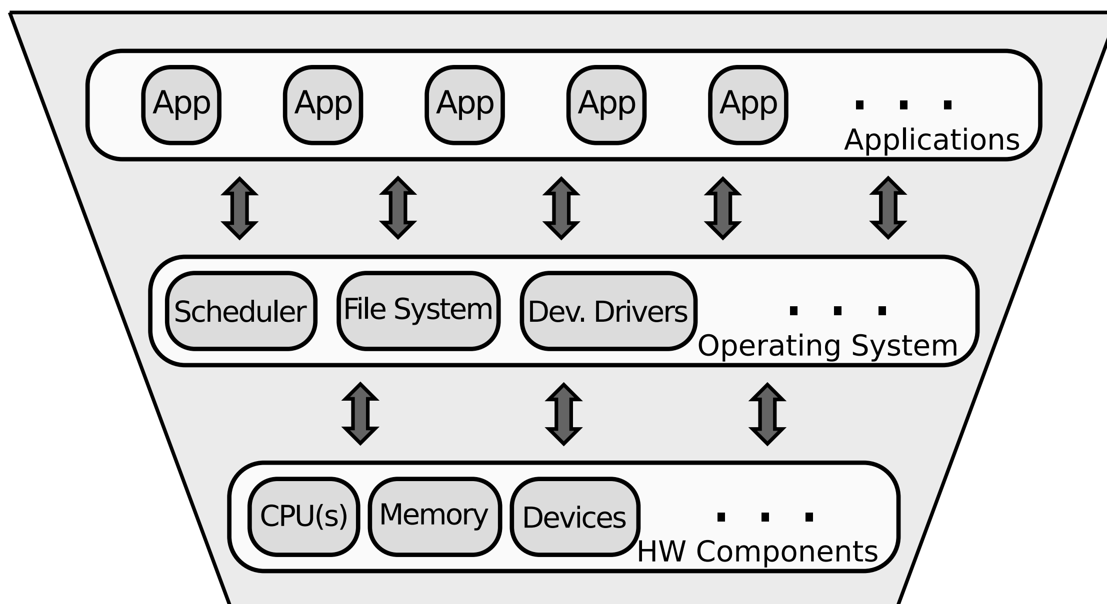


Figure 1.5: Main Layers in the structure of a modern computing system

some very specific task usually has limited hardware resources, a very thin OS layer and a few lightweight running applications, while grid [29] and cloud computing systems (e.g., Amazon Elastic Compute Cloud (EC2)[1]) may use resources spread across thousands of geographically distributed systems with a complex distributed OS and a huge number of computationally demanding applications. Even with all these differences, the basic three-layers model can be found in almost any modern computing system. According to this representation, it appears a reasonable and natural decision to choose the OS layer as the target for enabling self-adaptivity in a computing system. This is because the operating system layer already detains the role of managing the system resources and to enhance the system with self-adaptive properties, it makes sense to augment the operating system's management capabilities with autonomic features.

1.2.4 Adaptivity in Operating Systems

The goal of building autonomic computing systems is very generic and encompasses an entire field of research directed at realizing the vision outlined in Section 1.2. To be able to effectively realize this vision, though, it is necessary to identify a more specific area of research to be the

[1] http://aws.amazon.com/ec2/

first to benefit from the ideas of autonomic computing. Since almost any modern computing system relies on some sort of operating system (i.e., a software layer which exposes to users and applications higher level abstraction of the hardware interfaces), this is the component that has been identified by companies and researchers (IBM with K42 [8], the Massachusets Institute of Technology (MIT) with SEEC and Corey [38, 79] and others [14, 36, 42]) as the one where to introduce the autonomic paradigm in a computing system. This choice seems reasonable, since the OS is the component usually in charge of managing the system resources, so this is the system layer at which it is crucial to include autonomic features; this autonomic management layer could then exploit further autonomic facilities at the architectural level, if available.

The goal of building an autonomic (or self-adaptive) OS, translates to the creation of an operating system able to autonomously adapt the management of the underlying hardware resources in order to ensure the wanted performance to the running applications and to provide the best possible experience to its users, while taking in consideration the operating constraints (for instance, in terms of power consumption). In order to do so, the OS must be able to control itself by the means of the ODA loop and the following facilities must be implemented:

- System monitoring needs to be expanded to enable the OS to efficiently gather information about the status of its resources and other systems that may form the operating environment. This information is needed in order to perform the observation phase of the ODA loop and must include data on the working conditions of the system (for instance, the temperature of the processor(s), the available memory, ...) and on the current performance of the running applications.

- Decision policies must be implemented in order to analyze the data gathered by the system monitors and determine the proper actions to meet the current goals. In this context, also an efficient and user friendly mechanism for specifying goals must be provided to both end users and applications.

- Actuators are needed to alter the system status, thus providing the action phase. These actuators must be controlled by the decision policies and can act on any tunable parameter of the system; some common OS subsystems in which an actuator could be useful

are [38]:

- Frequency and voltage scaling, to adapt the computing power of the processors to the need of the running applications, avoiding at the same time to consume too much power or produce too much heat.
- Memory management, to provide the right amount and the right kind of memory according to application needs and system memory availability.
- Process scheduling and core allocation, to allow to each application a correct share of computing resources, in order to meet its goals while not interfering with other running processes.

The proposal of a methodology for enhancing a commodity operating system with autonomic capabilities and the implementation of support for applications' performance monitoring and adaptive process scheduling are the key contributions of this thesis for providing an enabling technology towards the creation of a complete autonomic OS.

1.3 Monitoring and Tracing

Monitoring and tracing techniques are useful, in autonomic computing systems, to perform the observation phase of the ODA control loop (or, equivalently, the monitoring phase of the MAPE-K or Self-adaptation control loops). Within this context, two types of monitoring may be identified [41]:

- *Passive monitoring* may be done just by using the existent operating system facilities to extract information about the system status. For instance, under Linux, the `/proc` pseudo-filesystem contains runtime information such as processor status, memory utilization, running programs, ...
- *Active monitoring* implies a modification of the monitored application (i.e., *instrumentation*) and/or of the operating system in order to retrieve some desired information which cannot be provided by the available system tools.

The monitor proposed in this thesis performs active monitoring, as it consists in a patch for the operating system and requires instrumenting the applications to be monitored.

Hardware and software monitoring components are not only used in autonomic systems, but are exploited for many different purposes (e.g., performance analysis, software and compilers optimization, fault detection, program characterization). Depending on the purpose they are used for, monitors may have different characteristics and provide different types of metrics. A rough classification of two different approaches towards monitoring defines the classes of *time-driven* versus *event-driver* monitors [39].

1.3.1 Time-Driver Monitoring

Probably, the most simple idea for characterizing a process (in our case, the execution of a program) is sampling its behavior through time. Within this context, two relevant concepts are the ones of *program state* and *execution trace* (Definitions 1.1 and 1.2).

Definition 1.1 (Program state)**.** *At any time t, the state $\Sigma_P(t)$ of an executing program P is a snapshot of its runtime properties (e.g., process counter, stack and head content, threads of execution, ...).*

Definition 1.2 (Execution trace [28])**.** *The execution trace of a program P is a sequence, possibly infinite, of program states $\Sigma_P(t_0), \Sigma_P(t_1), \ldots$.*

A time-driven monitor samples a relevant portion of the state of a program at regular intervals and, based on the obtained trace, provides summary statistical information about the program execution. This type of monitors is useful in some contexts, but they are insufficient for a behavioral analysis of the program [41], which is required for some applications.

1.3.2 Event-Driven Monitoring

A different approach to monitoring the execution of a program is based on the basic concept of *event* (Definition 1.3).

Definition 1.3 (Event [39])**.** *An* event *is an atomic instantaneous action.*

More specifically, an event may be a bit pattern in a register (when dealing with hardware monitoring) or a specific instruction which is inserted in the monitored program code in specific code blocks under investigation (program instrumentation). An event-driven monitor may record events in two manners, depending on the application:

- Creating an *event trace* which, similarly to the execution trace of a program, records a list of the events. Differently from the execution trace (which relies on regular time sampling), an event trace must contain, for each event, the timestamp at which it occurred.
- Storing, for each type of event, the number of times i occurred in an *event counter* (for instance, event counters may be found in most modern microprocessors [69]).

An event trace yields more information and allows more complex off line analysis [41]. If the monitoring information is to be used online (as it is the case within an autonomic control loop), however, an event counter may be sufficient.

This thesis proposes a software event-driven monitor which internally uses an event counter but also keeps a sampled trace of the values of the counter.

1.4 Process Scheduling

The monitor proposed in this thesis is used in an adaptive-scheduling autonomic control loop to build applications' performance awareness into the process scheduler of a commodity operating system. Scheduling is a generic problem and it is not constrained within OS research; this Section briefly introduces some theoretical issues about the generic scheduling problem and then focuses on operating system, providing some useful concepts about process scheduling.

1.4.1 Resource Constrained Scheduling Problem

The problem of scheduling or, more precisely, the *Resource Constrained Scheduling Problem (RCSP)* is a generic problem modeling a situation in which a set of *activities* must be completed by using a limited set of available *resources*, with the aim of optimizing one or more *objective function(s)*. More formally, a specific instance of the problem can be characterized by using a classification scheme of the form $\alpha|\beta|\gamma$ [63], where:

- α represents the resource environment.
- β indicates the activities characteristics (duration, precedence constraints, deadlines, ...).

- γ describes the objective function, which involves the maximization or minimization of a certain quantity subject to certain constraints depending on the context of the problem.

Once having defined the characteristics of the problem by using the proposed scheme, it is possible to give the following definition of the Resource Constrained Scheduling Problem:

Definition 1.4 (RCSP [63]). *Let J be a set of partially ordered activities and let $j_0, j_{n+1} \in J$ be a unique dummy beginning activity and a unique dummy terminating activity, respectively (so that always $J \neq \emptyset$). Let T be a set of temporal steps (e.g. years, weeks, μseconds, ...). Let $G(J, A)$ be an acyclic directed precedence graph representing precedence relations among the activities; i.e. $(j, j') \in A$ if and only if the activity j needs to be performed before the activity j'. Let R denote a set of resources and let c_{jr} be the processing time of the activity j over the resource r. Each activity j is to be assigned to exactly one resource r for being processed and that resource cannot process another activity $j' \neq j$ until j has been processed (i.e. after c_{jr} temporal steps).*
Under the above setup, the RCSP consists in minimizing the makespan, i.e. the maximum time necessary to complete all of the activities $j \in J$.

Definition 1.4 gives a generic definition of the RCSP, which applies to the case of process scheduling in operating systems, where the resources are constituted by the computational power of the system processor(s), the activities are the programs being executed and the objective is to assign the processor(s) to the executing programs according to given priorities among the programs. Specific terminology and concepts are in use in the particular field of process scheduling in operating system and are relevant to understand the state of the art and the contribution of this thesis; more details on this matter are provided in the next paragraphs.

1.4.2 Processes, Threads, Tasks

The problem of process scheduling arose in operating systems in the 1960s [72], with the need of running more than one program at the same time over limited computing resources (at the time, a single CPU). Within this context, it is useful to remember the following definitions:

Definition 1.5 (Process [72]). *A* process *is an abstraction of a program being executed, including its code, its data and all the information about its execution status.*

Definition 1.6 (Thread [72]). *A* thread of execution *(or, simply,* thread*) is a sub-entity within a process; it is a specific part of the executing program in charge of doing some precise elaboration. A process may be split into several threads which share the address space, open files and, in general, the resources assigned to the process.*

The former two definitions are quite classical and well-established, being defined by the Portable Operating System Interface for uniX (POSIX)[2] standard. Within this work, another concept is used, which is taken from the nomenclature used within the Linux kernel, to ease the discussion when talking about scheduling:

Definition 1.7 (Task [52]). *A* task *is a schedulable entity (either a process or a thread).*

Thus, from the point of view of the scheduler (at least, in Linux), processes and threads are perfectly equal and go under the name *tasks*. The following discussion uses the term *task* exactly as defined above, adapting as needed from the cited literature in case the term *process* was used with the generic connotation of *task*. An exception to this rule is the term *process scheduler*, which should be more consistently named *task scheduler*, but it has been chosen to keep the original name for historical reasons.

1.4.3 Multitasking

Another important concept related to process scheduling is specified in Definition 1.8.

Definition 1.8 (Multitasking [52, 72]). *The phenomenon of apparent contemporaneity of execution of several tasks on the same computer is referred to as* multitasking *and it is obtained by rapidly interleaving (as the process scheduler decides) the execution of the running programs on the available processor(s), thus giving the illusion of parallel execution.*

Historically, two different kinds of multitasking have been implemented in operating systems [52, 72]:

- In *cooperative multitasking*, the currently running task must explicitly yield the processor to let another runnable task (i.e. a task waiting for the CPU) be put in execution.

[2] Institute of Electrical and Electronics Engineers (IEEE) family of standards formally defined as *IEEE 1003*. The international name of such standard is ISO/IEC 9945.

- *Preemptive multitasking* gives more power to the scheduler, allowing it to *preempt* a running task, i.e. to suspend its execution in favor of another runnable task. The maximum time allowed for a task to run without being preempted is usually called *quantum* or *period*.

Cooperative multitasking was commonly used in operating systems up to Microsoft Windows 3.1 and Mac OS 9. This paradigm is simpler to implement (as it does not require any kind of time accounting or soft interrupts from the OS), but has a major disadvantage: one task could never voluntarily yield the CPU, leading all the other tasks to starvation (i.e. waiting forever for being executed). This could happen both if a malicious program is written in order to harm the system or if a program gets stuck in an infinite loop because of a bug. This drawback was historically addressed with the introduction of preemptive multitasking, which is the most common multitasking mechanism used in modern operating systems, including Linux and the more recent versions of Mac OSX and Windows.

1.4.4 Scheduling Environments and Goals

The process scheduler in an OS is in charge of solving the specific RCSP of allocating the computational power of the available CPUs to the runnable tasks in order to optimize a certain objective function. The nature of this objective function depends on the *scheduling environment* the system is operating in; a classification of the possible scheduling environments is proposed below [72]:

- *Batch* systems are characterized by a huge amount of jobs to be completed sequentially, without users impatiently waiting for interacting with a specific task. This environment is typical of servers or workstations for scientific computations

- *Interactive* systems have a certain number of users who want to interact with some tasks specific to each user. This is the typical case of desktop computers.

- *Real time* systems run tasks that need to respect specific deadlines for doing their job, so scheduling must be quite rigid to respect these constraints; these systems are further classified in:

  - *Hard real time* indicates that the deadlines expressed by the tasks are strict and the scheduler should return an error when not being able to enforce a deadline.
  - *Soft real time* schedulers, on the other hand, do not guarantee each deadline to be met, but usually guarantee a bounded Quality of Service (QoS) in terms of how many deadlines may be missed.

To highlight the differences between real time and non real time tasks, the latter are sometimes referred to as *best effort* [16], since there are no deadlines to enforce.

The classification of scheduling environments exposed above gives an idea of how the process scheduler objective may be different according to the context the OS is operating in. More specifically, some of the more common scheduling goals (which directly translate to objective functions) are reported below, associating them to the environments in which they are more relevant:

- Some scheduling goals are relevant in any environment [72]:
  - *Fairness* - giving to equal tasks (i.e. tasks with the same priority - or importance) a fair share of the CPU.
  - *Balance* - keeping all the parts of the system as busy as possible; spread the work load onto all the available resources.
- A batch environment targets the following goals [72]:
  - *Throughput maximization* - maximizing the number of completed jobs (i.e. tasks) per time unit.
  - *Turnaround time minimization* - making the time taken for a job to complete since it was first scheduled as small as possible.
  - *CPU use maximization* - keeping the processor always as busy as possible.
- In an interactive system, the scheduling goals are [72]:
  - *Response time minimization* - minimizing the time required for a task to respond to an interactive request from a user.
  - *Proportionality* - giving all the users the expected share of computing resources for their tasks.

- Real time systems have the following scheduling goals [72]:
  - *Meeting deadlines* - enforce the respect of the deadlines for real time tasks.
  - *Predictability* - being able to deterministically say in advance whether a deadline can be enforced or not.

The above classification seems very neat in assigning specific scheduling goals to each environment. The problem with real-world computing systems (e.g. a laptop or a personal computer) is that the actual scheduling environment is usually mutable according to the current running applications and to the current users' objectives. This fact makes it quite hard to determine the best scheduling goal for such systems, especially if this choice is to be made at compile time. An autonomic process scheduler should be able to go further and infer at runtime the current goal based on knowledge of the system status and on the performance goals expressed by the applications and user.

1.4.5 Task Classes

Another important element for the process scheduler to consider besides the type of environment (as described in Section 1.4.4) is the nature of the tasks that are currently competing for the use of the processor. A common classification of the behavior of task with respect to its interactivity is given below [52]:

- *Input/Output (I/O)-bound* tasks are characterized by frequently blocking (and thus voluntarily yielding the processor) waiting on I/O requests (for instance, writing on the disk or waiting for the user to type input data).
- *CPU-bound* tasks spend most of their time executing code.

Many process schedulers (e.g. the Linux O(1) scheduler [1]) tend to give more priority to I/O-bound tasks, with the aim of favoring interactivity (low response time), counting on the fact that these tasks run only for a short time and then voluntarily yield the processor. The drawback of such an approach is the difficulty of classifying a task as either I/O- or CPU-bound at runtime. Other schedulers (such as the Linux Completely Fair Scheduler (CFS) [45]) do not try to profile tasks to determine their interactivity level, but use different mechanisms to ensure good response time for interactive tasks.

1.4.6 Scheduling Policies

Since the introduction of multitasking systems, various simple scheduling algorithms have been designed. These algorithms are often tailored on one specific scheduling environment (see Section 1.4.4) and behave poorly when the environment changes. All the modern operating systems try to implement general purpose scheduling algorithm which should behave well under the real case of a mutable environment, hard to be classified in one specific category. The following discussion briefly presents some of the most known algorithms for each environment [72].

Scheduling in Batch Systems

In batch systems, the preferred scheduling algorithms are usually nonpreemptive or preemptive with a long time period. Two of the most known nonpreemptive scheduling algorithms suitable in batch systems are *First Come First Served (FCFS)* and *Shortest job First (SF)* [72].

FCFS is probably the simplest existing scheduling algorithm and it consists in placing the tasks in a First In First Out (FIFO) queue, assigning the processor sequentially to each task in the order of request. The simplicity of the algorithm is its main strength, but its noonpreemptive nature is a major disadvantage when both I/O- and CPU- bound tasks require the processor. In this context, the CPU-bound tasks will use much of the CPU time, thus sensibly slowing down the other tasks.

SF is a nonpreemptive algorithm that assumes that the run times for each tasks are known in advance. The rule is, again, fairly simple and it is to choose the shortest runnable job for execution. This algorithm, by always running the shortest task first, yields an optimal (i.e. minimal) turnaround time, provided that all the jobs are available simultaneously.

Scheduling in Interactive Systems

Interactive systems must prefer scheduling algorithms which favor interactivity, even if loosing performance in terms of throughput. Common scheduling algorithms in this context are *Round Robin (RR)* and *Priority Scheduling*, with the possible optimization of using *multiple queues* [72].

RR is a preemptive scheduling algorithm which introduces the concept of *quantum* of

execution time.

Definition 1.9 (Quantum [52]). *In process scheduling, a* quantum *(also referred to as* period*, or* timeslice*) is a specified time interval defining how long a task is allowed to continuously run before being preempted.*

The RR algorithm works by simply putting all the runnable tasks in a list and switching to the next runnable task whenever either the current task yields the processor or its quantum expires. The crucial parameter in RR is the length of the quantum, which should be tuned to get a proper trade-off between the required interactivity level and the efficiency in terms of throughput. In fact, using a short quantum ensures good interactivity, but it harms throughput due to the cost of context switch (i.e. the process of changing the task which is being executed). On the other hand, a long quantum allows better throughput but harms interactivity.

One of the assumptions behind the RR algorithm is that tasks are all equally important. It frequently happens, especially in multiuser systems, that not all tasks are equal and that some tasks, being more important, may need a greater share of CPU time than the other tasks. In such a context, the *Priority Scheduling* algorithm may help [72]. The idea behind this preemptive algorithm is that each task has a priority level, called *nice* level in UNIX-like operating systems; at preemption time (or when a task blocks), the runnable task with the highest priority is the next to be run. The base Priority Scheduling algorithm works with only one queue of tasks ordered according to their priority value.

In some cases, there is the need to adapt the duration of the quantum to the priority of the task (for instance, tasks at higher priority may be wanted to have a longer quantum than tasks at lower priority). In this cases, a *Multiple Queues Priority Scheduling* algorithm may be used [72]. This algorithm is an evolution of the Priority Scheduling algorithm and it works by using one queue per each priority level and moving tasks among the queues according to some kind of assumption about how interactive they will be in the near future.

Scheduling in real Time Systems

Scheduling in real time systems needs mechanisms that are quite different from the ones used for non real time systems. This is because in real time systems the tasks come with specific deadlines to respect and these are much more constraining on how to manage the

assignment of resources than throughput or interactivity goals. Usually, real time scheduling assumes that the tasks are periodic and each period is referred to as a *job* [72]. The main issue with scheduling in this context (especially in the case of *hard real time*) is that the scheduler needs to be able to predict whether it will be possible to meet each deadline and this is quite difficult to achieve, especially if the scheduling algorithm must be dynamic (i.e. it must make scheduling decisions at runtime). In order to do so, the behavior of each task (in particular, the worst-case execution time of each job) must be known to the scheduler, which thus needs a very precise knowledge of the tasks it is scheduling. Definition 1.10 more precisely defines a real time scheduling context.

Definition 1.10 (Real time tasking model [26])**.** *A real time system can be modeled as a set* $\Gamma = \{\tau_i\}$ *of* n *periodic tasks, where each task* τ_i *is modeled by a sequence of jobs, each described by the pair* (C_i, P_i)*:* C_i *is the worst-case execution time of the individual jobs and* P_i *is the minimum inter-arrival time between two consecutive jobs. An implicit deadline is that every job should terminate before the arrival of the next job.*

Within this context, one common concept (defined in Definition 1.11) is the one of *schedulability*:

Definition 1.11 (Schedulability [26])**.** *In a real time system, a set* $\Gamma = \{\tau_i\}$ *of* n *periodic tasks, where each task* τ_i *is assigned a computation budget of* Q_i *time units per reservation period* T_i*, is said to be schedulable if:*

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1$$

The assignment of Q_i *and* T_i *for all* $i \in [1, \ldots, n]$ *characterizes the different real time scheduling algorithms.*

Two of the most well-known ad classic process scheduling algorithms for real time systems are RMS and EDF:

- Rate Monotonic Scheduling (RMS) [50] is used when the priorities for the tasks are statically assigned based on the expected execution time of a job (the quicker the job, the higher the priority). This algorithm allows to pose a sufficient condition to the schedulability of n tasks as an upper bound on CPU utilization, which is about 70%, when $n \to \infty$ [50].

- Earliest Deadline First (EDF) [46] is a dynamic algorithm where the tasks are placed in a queue and, at each scheduling event, the task with the nearest deadline is the next to be executed. The schedulability condition for this algorithm is given by $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$ (see Definition 1.10 for definitions of C_i and P_i).

In general, real time scheduling algorithms are not suitable for scheduling ordinary tasks and usually (hard) real time operating systems offer a high-priority real time scheduling facility and non real time scheduling at lower priority for non real time tasks (for instance, RTLinux [83] provides a hard real time microkernel plus the ordinary Linux kernel, which is executed at lower priority and can be preempted at any time by real time tasks, to manage non real time tasks).

General Purpose Scheduling

The above discussion gives an idea on how complex the scenario of process scheduling in operating systems is, with its multiple environments, each requiring specific performance goals. This complexity makes it quite difficult to provide a general purpose scheduling algorithm with good performance in a mixed environment (with requirements in both throughput and interactivity). This is because it is hard to define an algorithm able to match these conflicting requirements (i.e., ensuring an acceptable QoS to the different running applications) in a continuously mutable environment such as the one generated in the normal use of a personal computer. Some schedulers have been able to achieve good general purpose performance either by using complex heuristics to analyze the running tasks at runtime (for instance, the Linux O(1) scheduler [1]) or by using fairness as the leading goal for scheduling decisions (this choice is the one behind the Linux CFS [45]). The breakthrough in this area, however, would be a scheduling algorithm able to know the tasks' goals in order to adapt its policy to give the best possible performance to each task according to its needs. One of the contributions of this thesis is the development of a process scheduler which, leveraging the ideas of autonomic computing and the ODA control loop, aims at achieving preliminary results in this direction.

1.5 Chapter Summary

An introduction to the concepts in the areas of autonomic computing, monitoring and tracing, and process scheduling, useful for the understanding of this thesis, was provided in this Chapter. The field of autonomic computing is quite recent and many researchers are working to realize the vision proposed at the beginning of this Chapter. Monitoring and tracing and process scheduling, on the other hand, are two less recent areas but both are related to autonomic computing systems: monitors are used in the autonomic control loop, while process scheduling is the OS component chosen in this thesis to be revised towards self-adaptation. The next Chapter contains an overview of some important works in autonomic computing, monitoring and process scheduling that are relevant to the scope of this thesis and form the state of the art which is the basis for the work presented in the subsequent Chapters.

Chapter 2

Related Works

An overview of some relevant works in the area of operating systems (with focus on OSes equipped with autonomic features) is proposed in this Chapter. The traditional design of OSes, originally conceived for classic single-processor architectures, is becoming outdated, being inadequate to support the new features introduced in modern architectures (e.g., multi- and many-cores or General Purpose computation on Graphics Processing Units); incremental changes to well-established OSes are not enough anymore and a need for a redesign is arising. Several projects are proposing innovation designing new OSes; a few of these are presented in Section 2.1. Some of the projects aimed at renewing the design of operating systems introduce autonomic features; these projects are particularly interesting and are treated more extensively in Section 2.3 (K42 by IBM) and Section 2.4 (Angstrom project by the MIT).

2.1 New Design Proposals for Operating Systems

The evolution of computing systems has led to a continuous increment of the their complexity. This trend is beginning to have consequences on how the operating system is able to manage the resources of the underlying hardware architecture [76]. The problem is that when the basic design at the base of the most diffused operating systems (e.g. Linux, Windows or Mac OSX) was conceived, the architecture of computing systems was much simpler than today. For instance, contemporary operating systems were originally designed to operate on a single core or a small number of cores and thus they are not well suited to manage systems with an increasing number of processors [76]. Another feature current operating systems are not

properly designed to manage is heterogeneity [56]. In fact, most contemporary commodity OSes are designed to manage systems that are uniform and cache coherent [56, 76], hypotheses that are starting not to hold any longer with the current evolution of system architectures. Moreover, the proposal, in 2001, of the autonomic computing paradigm posed new challenges to OS designers to address and alleviate the increasing difficulty of application developers to explicitly handle parallelism, energy efficiency, reliability and predictability issues [38]. To address the afore mentioned issues, several research projects are proposing new designs for the next generation of operating systems. Some of the most interesting projects are presented in this first part, highlighting the goals and the design characteristics of each:

- The Factored Operating System (fos) [76] (from MIT) is targeted at creating an highly scalable operating system for many-cores computing systems.
- Corey [79] is another project developed at MIT and it proposes a different management of data structures to reduce the performance footprint of the OS in multicore systems.
- Barrelfish [14] is developed at Eidgenoessische Technische Hochschule, Zurich (ETH, Zurich) in collaboration with Microsoft research at Cambridge and it proposes an operating system design referred to as the *multikernel*.
- Microsoft Helios [56] targets heterogeneous systems which can be managed with *satellite kernels*, an idea analogue to Barrelfish's multikernel.

Some more details about each of these projects are presented in the remaining of this Section, focusing on features that are interesting for this work; for a more complete presentation, refer to the cited literature.

The fos project starts from the observation that the trend in computer architectures is going in the direction of packing an increasing number of computing units in a single chip, leading to systems that, in the next decade, may reach 100s or even 1000s of cores per chip [2]. Since current operating systems were not designed for such highly parallel architectures (even if recent developments are highly improving scalability in this terms [15]), fos proposes a novel design to exploit the available parallelism, discarding the classical locking mechanisms and cache coherency requirements that hamper performance in this kind of scenarios. More

precisely, fos is designed according to a few recommendations proposed by the authors for future operating systems [76]:

- separate the execution resources of the operating system and the applications;
- avoid the use of hardware locks;
- avoid global cache coherent shared memory.

One of the most interesting proposals in fos is the separation of execution resources of the operating system and the applications. To realize this concept, in fos the system is decomposed in three main layers [76]:

- a thin microkernel;
- an OS layer, constituted by a set of servers which provide system services;
- applications which make use of the services according to their needs;

A portion of the microkernel executes on each processor core, in order to control access to hardware resources and to manage the delivery of messages (by the means of a name cache) for servers and applications communication. On top of this low-level layer, the system servers and the applications execute on separate cores, realizing the separation of resources between the OS and the applications. This approach is quite radical, as it requires the availability of a number of execution units in the system at least equal to the maximum number of executing threads. Thus, fos is designed for future many-core systems and current cloud systems [53, 77], which provide today the wide number of processing units required for the fos approach to be applicable. One very interesting consequence of the resource separation approach is that the process scheduler, in fos does not deal with time multiplexing, but treats a problem of space multiplexing. In fact, its duty is to define exclusive assignments of the cores, which are spatially distributed on the chip(s), to the tasks in execution. Other interesting points can be found in more recent publications [53], where two design principles were added by the authors to the recommendations mentioned above:

- The OS should adapt the use of resources to changing system needs, by measuring the utilization of each service and allocating more or less cores to its servers according to the current need.

- Faults in system services must be detected by a watchdog process and handled by the name server by reassigning faulted communication channels.

These two principles introduce some ideas of autonomic computing into fos, indicating that its design is being extended in this direction (see also Section 2.4).

Corey [79] is a prototype operating system that, as fos, addresses the computational efficiency problem on systems with many cores. The approach is less radical than the fos's one and the target systems for Corey do not require hundreds of cores: the focus is to make applications scale well with the number of cores by improving the management and exploitation of processor caches and shared data structures. The underlying observation is that with the parallel computing allowed by the presence of more than one core, contention on shared OS data becomes a problem and the applications spend a lot of time waiting to get the required locks, thus strongly hampering performance. The problem with contemporary operating systems is that shared data structures are required by the semantics of the OS to simplify resource management. Corey proposes the introduction of three new OS interfaces to improve the management of shared data, avoiding the use of shared structures unless strictly necessary [79]:

- *Address ranges* are a kernel-provided abstraction that corresponds to a range of virtual-to physical memory mappings. An application can define multiple address ranges, labeling each as shared or private. This gives more degrees of freedom in managing memory with respect to the two classic paradigms of having a unique address space (either private for single processes or shared among threads). This should allow to mark as shared only the data structures that really need sharing, reducing synchronization issues on data that are private to one thread.

- *Kernel cores* provide an abstraction that allows applications to dedicate specific cores to kernel functions and data. A kernel core can be dedicated to manage hardware devices or to execute system calls coming from applications being executed on different cores.

- *Shares* offer to the applications a means of dynamically create shared data (such as lookup tables) and defining at which level these data must be shared. This facility allows a finer control on shared structures, making it possible to avoid unnecessary contentions on data that need not be shared.

By introducing these interfaces, Corey is less radical than fos in that it does not enforce separation between OS and applications resources, but (with the kernel cores abstraction) it gives to applications the possibility of choosing whether to require a separated execution environment for a particular OS service. This makes Corey more flexible than fos and allows it to bring performance improvements starting from systems with eight or more cores, as shown in experimental results [79]. These interfaces should enable a finer control on the sharing of resources so that subsystems as the process scheduler (which often uses shared data, hampering performance with many cores [79]) can be made more efficient and scalable. The drawback of such approach is that the system relies on a wise use of these interfaces by the applications to ensure good performance, thus exposing even more complexity to application developers. The approach taken by Corey is interesting in addressing efficient exploitation of new hardware architectures but, probably, it could be more effective if it provided autonomic features to hide some complexity from the application developers.

The Barrelfish [14] operating system is another project addressing the scalability problems of operating systems on new multicore and heterogeneous architectures and it does so by proposing to rethink the structure of an OS as a distributed system of functional units communicating by means of explicit message passing. Barrelfish developers aim at creating an operating system able to flawlessly scale to heterogeneous architectures. The peculiarity of such architectures is that they often contain programmable units which cannot be made cache coherent with the rest of the system (e.g. GPUs or NICs) or do not support shared memory at all. To be more precise, a heterogeneous system can be characterized by at least three types of diversity [65]:

- *Non-uniformity* refers to non uniform memory architectures (e.g. Non Uniform Memory Access (NUMA) systems); this characteristic will increase its relevance in future architectures, making the structure of processing units resemble a network.

- *Core diversity* is not very diffused today, where most of the multiprocessor architectures are homogeneous (i.e. all of the processors are exactly the same), but it is normal to find it in embedded systems, where there are often different specialized processing units. Moreover, core diversity will become an issue in any system with the increasing practice of exploiting GPUs or FPGAs for specific tasks.

- *System diversity* indicates diversity among the hardware components of different systems. This diversity is very evident (think of a mobile phone against an internet server) and it makes it difficult for application programmers to write software required to run efficiently on diverse platforms.

To address these three levels of diversity, the developers propose a design of a system where all inter-core communications are explicit; this model is called *multikernel* [14] and it is represented in Figure 2.1. The *multikernel* defines the design of an operating system formed

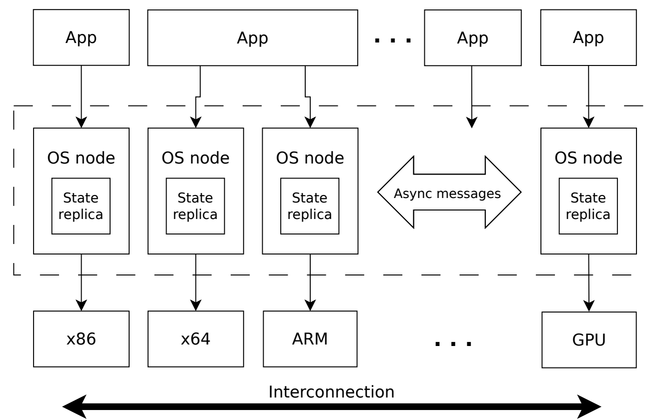


Figure 2.1: Multikernel model structure (adapted from [14])

by separate entities in execution on each programmable component of the system and asynchronously communicating by means of message passing. This structure makes the overall OS hardware-neutral, in that only the OS nodes contain architecture-dependent code, but the operating system as a whole is unaware of the specific architectures of the processors it is running on. One interesting part of this model defines the design of a process scheduler based on five design principles that are thought to be important for supporting the mixed workload expected on a general-purpose multicore or heterogeneous system [58]:

- Time multiplexing on each core is still needed and scheduling is not to be reduced to spatial partitioning. This is particularly important in heterogeneous architectures, where the computational power is not equal among the different processing units.

- Scheduling is to be performed at different time scales (i.e. at different frequencies): *long-term* placement of applications onto cores, *medium-term* resource allocation in reaction to applications demands and *short-term* fine-grained per-core thread scheduling.

- The scheduler should reason online about the hardware to use the best policy on each node. This is implemented in Barrelfish through a System Knowledge Base (SKB) based on a subset of First Order Logic (FOL) where information about the hardware is maintained.

- Online reasoning should be applied to each application. This implies that the scheduler should know about applications workload and requirements; to enable this, Barrelfish allows applications to present a *scheduling manifest* exposing long-term requirements as constrained cost-functions in the ECRC Common Logic Programming System (ECLiPSe) language [17].

- Applications and OS must communicate to negotiate resource allocation. This is done by using *dispatcher groups* that should allow performance tuning according to the applications' needs.

The first principle sharply distinguishes the Barrelfish approach from the ideas in fos, where time multiplexing is completely dropped, relying on the abundance of homogeneous cores. The other four principles introduce concepts affine with the autonomic computing ideas, with the use of a System Knowledge Base used to make the OS aware of its status. As for the interfaces exposed by Corey, the pitfall with this approach could lie in relying too much on applications programmers, who are in charge of expressing scheduling goals in a manifest encoded in a (possibly) unfamiliar logic language.

Helios [56] is an operating system which shares many goals and design features with Barrelfish. More precisely, Helios targets heterogeneous platforms, with the goal of simplifying the task of application development on such systems. To achieve this goal, Helios is based on *satellite kernels*, which define a model very similar to the multikernel proposed in Barrelfish, where the satellite kernels correspond to the multikernel OS nodes. A satellite kernel can run on any programmable component featuring:

- a CPU;

- some amount of Random Access Memory (RAM);

- a timer;

- an interrupt controller;
- the capability of catching exceptions (i.e. trap).

These requirements are quite constraining with respect to current hardware components: for instance, GPUs are generally not equipped with timers or interrupt controllers, thus preventing a satellite kernel to run on such a component. With respect to this, the authors state that these requirements will be widely satisfied in the next generation of hardware components. Satellite kernels are designed to export a single, uniform set of OS abstractions across the whole system. The duty of satellite kernels is to hide the heterogeneity of the system to the application programmer, who can rely on the Application Programming Interface (API) and abstractions offered by the OS. Each satellite kernel is a microkernel composed of a scheduler, a memory manager, a namespace manager and a module in duty of managing communication with the other satellite kernels. The communication between applications or services happens by the means of a message passing system, which is implemented for both communications within the same satellite kernel and for communications between different satellite kernels. The capability of exposing a single consistent system-wide abstraction is made possible thanks to this mechanism of message passing, which is split in two parts [56]:

- *Local message passing* manages communications within a single satellite kernel.
- *Remote message passing* is used when a communication must traverse between different satellite kernels.

The local message passing system is provided by the Singularity framework [42], over which Helios is built. This framework supports safe and efficient process software isolation (Software Isolated Processes (SIPs)) and a fast *zero-copy* means of passing messages within the same address space. Thus, local message passing (which should be the most common case of communication within the OS) has a very fast implementation. Remote message passing is made possible by the existence of a system-wide namespace; it is less efficient (it requires copying data), but it allows to maintain the same communication style even when dealing with inter-kernel communications. To exploit this unique system abstraction over heterogeneous hardware, the Instruction Set Architectures (ISAs) of the disparate processing units hosting the satellite kernels must be encapsulated and a unique programming language must

be available for application development. This is achieved in Helios by using a two-phase compilation strategy, where an application written in Sing# (which is a derivative of the C# language [32]) is first compiled into an intermediate language - the Common Intermediate Language (CIL) of the .NET framework - and then translated to the ISA of the unity where it is to be executed. An application can express preferences about the type of device onto which be executed. This is made possible by allowing the specification of an *affinity* value with another process (e.g. a device driver or a satellite kernel), which can be positive (indicating preference for being executed on the same unit) or negative (to indicate preference for separated execution).

2.2 New Operating Systems and Autonomic Features

The projects afore illustrated in the previous Section propose ideas and design principles to renew operating systems in order to get the most out of the new hardware architectures that are becoming available. The main effort is to better support multicore and heterogeneous systems, which are the big novelties in the new architectures with respect to those contemporary operating systems were designed for. Some projects - i.e. fos and Corey - are more targeted towards multicore systems, while others - i.e. Barrelfish and Helios - are more explicitly aimed at supporting heterogeneous architectures. Beyond the different realizations, there are some common ideas among these projects, which will probably be at the base of commercial future operating systems:

- The shared memory model is problematic with multicore and heterogeneous architectures. New operating systems should base internal communication on the message passing paradigm.
- Especially in heterogeneous systems, the OS should consist of a network of small nodes, one for each programmable computing device available in the system.
- The key feature in the operating systems of the new generation will be the ability to scale very efficiently with respect to the availability of (heterogeneous) computing resources.

These ideas can lead to a great innovation in operating systems design, but the core of these projects does not thoroughly take into consideration the issues in terms of complexity for users

and applications developers (see Chapter 1). Other projects exist that are more explicitly trying to innovate by founding future operating systems on the autonomic paradigm. These projects share with the already mentioned ones some design strategies, but are more targeted towards enabling self-* properties (Section 1.2.1) in a newly designed OS. Two interesting projects with these characteristics have been analyzed:

- K42 [8] is the proposal of IBM research to address the need of autonomic capabilities in computing systems [61].
- The Angstrom project [75] is being developed at the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL) and it is intended at creating a fundamentally new computing architecture to meet the challenges of extreme-scale computing. The Angstrom project is aimed at extending fos with the autonomic featured provided by a SElf-awarE Computational model (SEEC) [38], thus realizing a SElf-aware Factored Operating System (Sefos).

An analysis of the K42 operating system is proposed in Section 2.3, while SEEC and Sefos are presented in Section 2.4.

2.3 The K42 Operating System

K42 [8] is a research kernel designed for cache-coherent 64-bit multiprocessor and NUMA systems and it is based on the Tornado operating system [33]. The key goals of K42 include [8]:

- Efficiently scale both up towards large multiprocessor and NUMA systems and down to small multiprocessors, running as efficiently as kernels that do not scale up.
- Be modular and simple extensible, being available as open-source software to a large research community.
- Allow applications to customize the OS behavior in how it manages their resources and let the system adapt to changing workload characteristics.

To achieve these goals, the overall structure of K42 is based on a microkernel design [7], where there is a small exception-handling component (the microkernel) and a number of

servers which marshal all of the operating system functionalities. The microkernel provides basic functionalities such as memory management, process management, Inter-Process Communication (IPC), networking and device support (where the latter two are planned to be moved out of the microkernel [8]), while the servers provide all the more advanced OS functions (e.g. file system, sockets, pipes, . . .). Each server lives in its own separate address space and the system relies on a fast IPC mechanism to allow efficient communications among the servers and from the applications to the servers. Moreover, K42 moves some functionalities traditionally implemented in the kernel (e.g. thread scheduling) to userspace libraries, allowing application developers to redefine the behavior of such modules. This system structure is all built upon object orientation (the OS code is written in C++), allowing a high level of modularity, but still maintaining API and ABI compatibility (by the means of an emulation layer) with programs written and compiled for Linux.

2.3.1 Autonomic Capabilities Through Hot-Swapping

The object oriented design of the kernel allows K42 to support online reconfiguration [67] and dynamic update [13] mechanisms, which permit to modify (i.e. alter or substitute) the components of the OS and to apply updates to the system without any downtime. These mechanisms are realized through a procedure called *hot-swapping*, which consists in allowing monitoring code, diagnostic code and function implementations to be dynamically inserted and removed in live systems [4, 5]. This ability of supporting interposition and replacement of active OS code (both at server and at userspace-library level) allows K42 to show autonomic capacities of self-adaptation to a changing environment in several ways [5]:

- The system can be highly optimized for the common case, while ad-hoc solutions for the uncommon cases may be hot-swapped at need.
- Caches and memory management policies can be swapped at runtime according to the current data access pattern, in order to always use the best performing policy with respect to the current status.
- Support for architecture-specific features can be hot-swapped, allowing to exploit all the features peculiar to each architecture but keeping a fast and simple architecture-independent base implementation.

- Shared and partitioned version of the file pages caching mechanism are hot-swappable, allowing optimization for sequential or highly parallel applications.
- Applications can provide specific implementations of the OS services, which are implemented as userspace libraries, and K42 is able to hot-swap to the application-provided implementation as needed.
- Monitoring objects can be interposed into the relevant OS code sections by the applications that benefit from the information gathered by such monitors. Applications that do not require monitoring will thus not be slowed down by the execution of the monitoring code.

To effectively perform these operations in an autonomic way, a very important feature in any autonomic system is the monitoring system which must provide updated and accurate information about the current status of the system. Further discussion of the K42's (and others) monitoring systems can be found in Section 3.1.3.

2.3.2 Process Scheduling

The process scheduler in K42 is based on an interesting approach named *two-level scheduling* [3]. The main feature of this approach is the division of the process scheduler in two subsystems: one running in kernelspace and one in userspace. Moving part of the process scheduler to user level is quite unconventional but it helps in two directions [3]:

- The userspace scheduler subsystem operates on single threads without the need of context switches and without the kernel even being aware of what its operation. This can improve performance in many situations.
- The partial implementation of the scheduler in a userspace library allows the applications to tailor, if needed, this scheduling level at their own needs, simply by reimplementing the library.

As said, the user-level scheduler treats threads; more precisely, it schedules threads that live within the same address space (i.e. that belong to the same process) and are grouped in an entity called *dispatcher*. The threads contained in the same dispatcher are indistinguishable

to the kernel-level scheduler and are fully managed by the user-level scheduler, which usually does its job without any need of context switching. The kernel-level scheduler treats dispatchers and it assigns the resources by using entities at a higher level in the hierarchy, called *resource domains*. Each resource domain groups a subset of the dispatchers and it is the entity which actually owns the rights to use the hardware resources. The hierarchy just described is represented in Figure 2.2 in the case of a system with four processors. Each resource domain

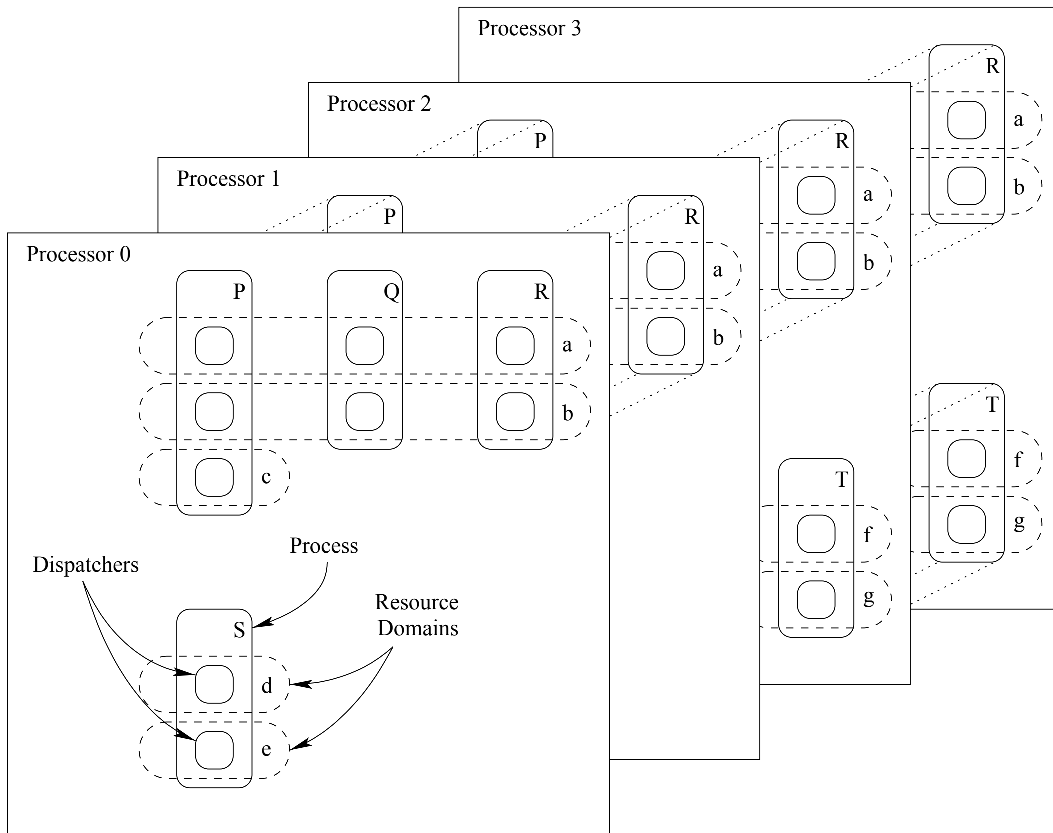


Figure 2.2: Scheduling entities in K42 [3]: dispatchers, processes and resource domains

owns a fraction of each available CPU and the kernel-level scheduler is in charge of fairly assigning each processor to the resource domains according to its owned fraction. The expected use for resource domains is to fairly assign CPU time to users by binding one resource domain to each user. Within a resource domain, each dispatcher is bound to a specific CPU and the kernel may move a dispatcher to a different processor for load-balancing purposes.

This two-level scheduling approach has the advantage to decouple the assignment of CPU time (which is done at the granularity level of the dispatchers) and the scheduling needs

of the applications. A process could use a single dispatcher with many threads and define the scheduling policy that marshals the threads by reimplementing the user-level scheduling code; this would be equivalent, at the kernel-level scheduler, to a single-threaded process. On the other side, a process could use multiple dispatchers to attain real parallelism (on a multiprocessor system) or to assign different scheduling characteristics (e.g. priority) to different sets of threads. This mechanism is indeed quite powerful but, as for other techniques illustrated before (e.g. in Corey and Barrelfish - presented in Section 2.1), it strongly relies on application developers, requiring - in order to have maximum control - to completely redefine the user-level scheduling policy.

2.4 Angstrom Project

The Angstrom project [75] has been recently created to deal with autonomic systems and it is currently targeted at extending fos [76] in an autonomic direction. As illustrated in Section 2.1, fos implements the operating system services (for instance the file system) as *fleets* of servers bound to some cores in a multicore or cloud system. The goal of the Angstrom project is to create a new service in charge of providing fos with autonomic capabilities. The framework for realizing this layer is called SEEC [38] and it presents the typical structure of an autonomic system based on the ODA control loop; more precisely, the SEEC framework is composed of three main components [38]:

- an observation layer;
- a decision layer;
- services in charge of taking actions on the system.

As in the classic ODA loop, in SEEC the observation layer is in charge of monitoring the status of the system and of providing this information to the decision layer; the duty of the decision layer is to analyze the available information in order to determine what actions should be taken and to trigger the action of the services on the system. Currently, the only monitoring interface available in SEEC is an API named Application Heartbeats (AH) (further investigated in Section 3.1.3). The decision layer is realized by a control theory-based control

system which supports three controllers with different dynamic behaviors. Finally, there exist some services acting on various parameters of the system [38]:

- A frequency scaler uses Dynamic Voltage and Frequency Scaling (DVFS) to adjust the clock speed of the system processor(s).
- A core allocator is in charge of assigning a subset of the available system processors to the running processes.
- A Dynamic Random Access Memory (DRAM) allocator is useful when more than one memory controller is available on the system (which is true, for instance, on NUMA computers) and it is capable of assigning the memory controllers to the running processes.
- A power manager uses combined actions of the previous services (which, in turn, determine higher or lower power consumption) to directly affect the power consumption of the system.

Currently, the SEEC framework has not yet been implemented as a fos service, but has been implemented over Linux. This implementation is completely done in userspace [38], avoiding the need of directly modifying the kernel, but it posing some limitations on the effectiveness of the autonomic actions. For instance, the *core allocator* service, which is in charge of assigning a subset of the available system processors to the running applications, acts on the system by altering the *affinity mask* of a process. In Linux, this value is defined for each process and it allows to indicate a subset of the available cores over which the process may be scheduled; the core allocator changes the affinity mask from userspace through a system call (*syscall*), incurring in the overhead due to the kernel crossing. Experimental results [38] show that this strategy can yield performance improvements, but the realization of such service in userspace poses a number of limitations (beyond the overhead due to system calls) on its precision in affecting the system status. The main limitation is that the core allocator has the only chance of modifying the affinity mask to direct the scheduling of the process, but the final decisions are taken by the Linux scheduler, which autonomously determines, for instance, whether a task is to be moved for load-balancing reasons. This example shows how working in userspace can be limiting in a monolithic kernel such as Linux, allowing just a loose control on the

system. This consideration is one of the reason underlying the choice of directly patching the Linux kernel for the implementation of the monitor and adaptive scheduling mechanism proposed in this thesis.

2.5 Summary

The works presented in this Chapter rise interesting issues in the ability of contemporary operating systems to really fulfill their role of exposing efficient and simple interfaces to ease applications developers in accessing all the resources offered by the bare hardware. These works on innovation in operating systems suggest that new approaches are needed to successfully exploit the new available architectures and the ideas of autonomic computing can be the base over which building a research effort to address these problems.

Chapter 3

State of the Art in Monitoring and Scheduling

The work proposed in this thesis has the goal of providing a methodology for enhancing a commodity OS with autonomic capabilities leveraging the ODA loop. To do so, both a generic monitoring infrastructure (to provide the observation phase) and an adaptive process scheduler (as a proof of concept realization of the decision and action phases) are proposed. To help the design of this components, a preliminary study has been conducted on previous works on monitoring and process scheduling non necessarily related with autonomic computing. Some of these works are illustrated in ths Chapter: Section 3.1 covers related works dealing with monitoring and Sections 3.2 to 3.4 present existing process schedulers.

3.1 Runtime Monitoring Infrastructures

Gathering relevant runtime information is a crucial operation in the effort of building autonomic computing systems. For this reason, an autonomic system must be equipped with a proper monitoring infrastructure able to provide the information required to enable the wanted self-* properties. Monitoring infrastructures in computing systems may be built in different forms and work at different levels; this Section provides an overview of some interesting works available in literature which were considered during the design of the monitor proposed in this thesis.

3.1.1 Low Level Processor Monitors

Most modern general purpose processors are equipped with on-chip hardware giving runtime information. Performance monitoring hardware in processors is characterized by two components: *performance event detectors* and *event counters* [69]. The information provided by these hardware monitors represents low-level details on the efficiency of the processor in executing the code; for instance, there exist event counters for the number of completed instructions of a certain type (e.g., floating point) or for characterizing branch prediction (e.g., number of mispredicted branches) [69].

Since the data provided by hardware event counters represent very low-level information, they are not easy to be directly used to infer useful information about the running programs. To allow the use of this information, researchers proposed APIs that ease the access to hardware monitors. Within the High Performance Computing (HPC) community, Browne et al. [18] proposed - back in 2000 - a common interface towards performance counters able to support different microprocessors and operating systems (from the Intel Pentium Pro/II/III on Linux to the Cray T3E,EV5 on Unicos/mk). The monitoring hardware, however, became much more complex with the next generation of microprocessors and , in 2004, Sprunt [68] proposed two tools, called *brink* and *abyss*, to provide a high level simple interface to the complex performance monitoring hardware found on Pentium 4 processors. This work represents one of the first efforts to exposing high level interfaces - through descriptions in the eXtensible Markup Language (XML) - to the complex hardware event counters of those processors. More recent works consider the fact that performance counters in microprocessors were not designed to be exploited by the users, but simply as debugging tools for hardware architects [78]. One proposal for overcoming the limitations posed by traditional performance counters is to introduce reconfigurable or programmable monitoring elements within the hardware architecture, able to decouple the monitoring action from the operation of the CPUs [64]. A different approach has been proposed by West et al. [78], who focus on monitoring multicore processors and propose the introduction of a specialized CPU core dedicated at the collection ad evaluation of performance data.

The low level processor monitors are improving from just providing counters for some events (which are hard to use other than for low-level debugging) to allowing more flexible and

structured access to specific runtime information. Recent works propose architectural changes (such as the introduction of a dedicated core [78] or the use of reconfigurable components [64]) in order to extend the utility this approach from debugging and statistical analysis to runtime use of monitoring information for improving performance and hardware utilization by the software being executed.

3.1.2 Software and Applications Monitors

A different approach, which does not require modifications in the hardware architectures, to provide useful application-specific information at an higher level than hardware counters is operating at software level. Software monitoring usually consists in instrumenting (either manually or automatically) the applications to be monitored by introducing calls to a monitoring API. This kind of runtime software monitors are used for profiling, performance analysis, software optimization as well as software fault-detection, diagnosis, and recovery [28]. An interesting idea in this context which has been picked up in the work proposed by this thesis is the use of the *procfs* pseudo filesystem to expose monitoring information (as done by Jancic et al. [44] for monitoring cluster applications).

In fault detection software monitors, a well established practice is the use of “I-am-alive” signals called *heartbeats* [70]. In fault detection applications, these signals are used to asses the correct operation of a specific component; the same idea has been later used, in a self-awareness context different from fault detection, for the purpose of measuring applications' progress [37] (see Section 3.1.3). This idea of heartbeats as a generic monitoring unit for measuring progress is at the base of the monitor proposed in this thesis.

3.1.3 Monitoring in Autonomic Computing

Two notable examples of the use of monitoring infrastructures in a self-aware and autonomic context are given by the K42's performance monitoring and tracing infrastructure [6] and by the monitoring facility used in the SEEC framework [37].

Performance Monitoring and Tracing in K42

K42 is equipped with a tracing infrastructure that manages the logging of any interesting system event for debugging or monitoring of the system [80]. The key features of such infras-

tructure are [6]:

- The provision of a unified set of events for any monitoring activity, including correctness debugging, performance debugging and performance monitoring.
- When not in use, the impact of the monitoring infrastructure on system performance is low enough to allow it to be kept compiled-in, permitting data gathering to be dynamically enabled/disabled at runtime. It is also possible to completely exclude (at compile time) some events from being built to obtain zero impact on useless events.
- The monitoring infrastructure takes care only of collecting and making available the events; the analysis of such information is thus decoupled from its gathering.
- The event logging mechanism is flexible enough to provide cheap collection of data for both small or large amounts of data per event.

These characteristics, coupled with the hot-swapping ability to insert monitoring objects into the operating system code at runtime, give to K42 a suitable way of gathering information on its state, which is a crucial capability for an autonomic system. More in details, the monitoring infrastructure is used in K42 to realize a Continuous Program Optimization (CPO) [19, 81] paradigm, where the information provided by a Performance and Environment Monitoring (PEM) infrastructure are used by CPO agents to address performance problems. The information provided by the PEM come under the shape of XML-specified events (from low level events, such as cache misses, to higher level software related events) and the CPO infrastructure provides an API for manual instrumentation.

On one hand, the proposed infrastructure is complete and potentially supports multiple programming languages. On the other hand, PEM poses a notable burden on systems and applications developers, who must provide both an XML specification of each event and the associated code.

Application Heartbeats

The monitoring facility found in literature which is most related with the work presented in this thesis is part of the SEEC framework used in the Angstrom project (see section 2.4) and is called *Application Heartbeats* [37]. Application Heartbeats is a monitoring API providing

a simple way of measuring applications' performance in critical sections by relying on the abstraction of an *heartbeat*. Within this context, a heartbeat is a periodic signal sent from the application to the API to indicate progress in its job. The metric used by Application Heartbeats to evaluate the current performance of the monitored applications is their current *heart rate*, i.e. the number of heartbeats each application has sent in a time unit. Obviously, the heart rate of an application is measured in $\frac{heartbeats}{second}$.

Application Heartbeats allows the applications to specify a target heart rate and a target latency between specially tagged heartbeats: once the application has set its targets, it can send heartbeats to enable the monitoring of its current performance. An example of use of this monitoring API is a video encoder [37] that sets a target heart rate of $30\frac{heartbeats}{second}$ and sends one heartbeat after each encoded video frame. In this way, the application indicates the will to be assigned enough resources to produce thirty frames per second. In fact, this kind of application has the right behavior to be monitored with the approach of sending heartbeats; i.e. there is a computational intensive code section, which is realized as a loop, and the straightforward way of measuring performance is to send one heartbeat at each iteration of the loop. Unfortunately, not all the applications have a behavior similar to this and thus not every application can be monitored with this approach. Even if limited to a specific class of applications the monitoring offered by Application Heartbeats is quite effective and it has the great advantage of being simple, requiring a relatively small effort from the application programmer.

3.2 Process Scheduling in Linux

Some of the projects presented in the first part of this Chapter (K42 in particular - see Section 2.3.2) propose ideas for the process scheduler to better support the new architectures they target. The process scheduler is a central component of most modern operating systems and it is often target of continuous evolution aimed at better supporting all of the working conditions that the system must support. In this context, an interesting case study is offered by the evolution of the process scheduler in the Linux kernel. The information to analyze the evolution of the Linux process scheduler is easily retrievable thanks to the open source and community based development model of this kernel: this model encourages contributors to

question the existing implementation on the public mailing list and to freely propose patches, leading to continuous evolution [52]. This Section provides a brief history of the evolution of the process scheduler (for non real time tasks) in Linux and a presentation of how this component works in the current versions of the kernel. The current process scheduler used in Linux provides the base for the implementation of the adaptive, performance aware scheduler proposed in this thesis.

3.2.1 $O(1)$ scheduler

The initial versions of the Linux kernel (the first version was published by Linus Torvalds in 1991 [52]) did not focus on having a very efficient process scheduler and, up to the 2.6 series (more precisely, up to Linux 2.6.8.1, released in 2004 [1]), this component was very simple and scaled quite poorly when increasing the number of processes or number of available processors [52]. The first interesting implementation of the Linux scheduler is was written by the developer Ingo Molnar and merged in the Linux mainline sources with the release of version 2.6.8.1 [1]. This process scheduler takes into consideration the heterogeneous targets of Linux, which is used in both servers and desktop systems (which usually have different scheduling goals, as explained in Section 1.4.4). One of the most interesting features of this scheduler is that it is guaranteed to do its job (i.e. picking the next task to be executed) in a constant time, not depending on the number of tasks to be scheduled. An algorithm with this characteristic is denoted as $O(1)$, with the Big-O notation [23]; hence, this implementation of the Linux scheduler is commonly referred to as the "$O(1)$ *scheduler*". The capability of this scheduler to always run in constant time allows it to scale very well with the number of tasks and it is attained thanks to two key data structures [1]:

- For each available processor, there is *runqueue*, which contains all the runnable tasks assigned to that CPU.
- Each runqueue contains two *priority arrays*, respectively marked as *active* and *expired*. The tasks are moved to the expired array as they run out of their timeslice (see Definition 1.9) and, when no more tasks remain in the active array, the scheduler simply inverts the labels (which entails simply updating two pointers, which is done in constant time).

Priority arrays contain linked lists, one for each possible priority level, which enumerate all the runnable tasks for the associated priority; each task has an assigned timeslice which is computed when the task is moved from the active to the expired array. The priority assigned to the task depends on a *static* value (called *nice* value, according to the POSIX standard) and on a dynamic component which is computed by the scheduler to improve interactivity. To do so, the $O(1)$ scheduler uses a quite complex [45] heuristic to classify the tasks as CPU-bound or I/O-bound and gives higher priority to the tasks classified as I/O-bound, which are identified as the ones requiring better interactivity. Moreover, this scheduler supports load-balancing mechanisms to keep the workload distributed on the available processors and offers a very simple soft real time support managing real time tasks with FIFO or RR queues at maximum priority.

3.2.2 Completely Fair Scheduler

The scheduling algorithm currently used in Linux is called CFS and it has been introduced in the kernel version 2.6.23, released in 2007 [52]. This scheduler was designed by the same developer who wrote the $O(1)$ scheduler and it is intended for resolving the limitations of its predecessor, mainly linked with the complex heuristics used to determine the interactive or batch behavior of the tasks [45].

The design of CFS has been influenced by the Rotating Staircase Deadline Scheduler (RDSL), proposed by the kernel hacker Con Kolivas but not merged into the Linux mainline sources. RDSL was based on the idea of being fair in CPU assignment, without trying to characterize the behavior of the tasks [48] and CFS, as the name suggests, fully embraced this idea. To achieve fairness, in CFS, the classic concept of timeslice (also used in the $O(1)$ scheduler) was discarded and a the idea of *virtual runtime* of a task (Definition 3.1) was introduced.

Definition 3.1 (Virtual runtime [52])**.** *The* virtual runtime *(or, in short,* vruntime*) of a task is the actual runtime (the amount of time spent running) of the task normalized (i.e. weighted) by the number of runnable tasks. It is measured in nanoseconds.*

The idea behind the virtual runtime is similar to the concept of Virtual Finishing Time (VFT), which was in use in fair queuing algorithms as a way to measure the degree to which an

activity has received its proportional allocation of resources [54]. The vruntime, specializing this concept to process scheduling, represents the time a task would have run on an ideal machine able to support perfect parallel execution (i.e. with as many processors as the number of runnable tasks).

The virtual runtime of the tasks being executed is updated either at each *scheduler tick* (which, in CFS is dynamically determined and not fixed) or when a task yields the processor. The nice value of the running tasks is taken into account during the vruntime update operation and it is used to weigh the update value: tasks with a higher priority (lower nice value) will get a larger weight and a smaller vruntime update. Thanks to this mechanism, the CFS does not have any concept of a fixed timeslice and the scheduler simply chooses the task with the lowest vruntime [34], leaving it in execution as long as it has the lowest virtual runtime within its runqueue.

As in the $O(1)$ scheduler, in the CFS there is one runqueue for each available processor; the difference lies in how this runqueue is implemented. In CFS, the runqueues are based on *red-black trees*, which are a particular class of balanced binary trees [23]. This data structure allows insertion and deletion complexity of $O(\log n)$, where n is the number of nodes (i.e. the number of tasks in the runqueue) and it is topologically ordered so that the node with the minimum index (i.e. the task with the minimum runtime) will always be the leftmost leaf of the tree [52].

Thanks to its focus on fairness, the CFS achieves good scheduling performance with respect to both interactivity and throughput (sometimes sacrificing the latter for maintaining the former [34]). Part of the contribute of this thesis (see Section 5.3) consists in defining a way of enhancing the CFS with performance awareness, making it take into account the current and the declared target performance of the running applications and consequently adapt its choices with the aim of satisfying the needs of the applications.

3.3 Real Rime Schedulers

Scheduling real time tasks traditionally defines a different context with respect to scheduling non real time tasks. In fact, real time scheduling is usually required in specific contexts, with well defined requirements of deterministic execution times, such as in automatic

controllers [30], embedded systems, or multimedia applications [25]. In modern computing systems, heterogeneous scheduling needs of different tasks running in parallel may blur the differences between the traditional scheduling environments [16] (see Section 1.4.4). Real time tasks, however, are well separated from non real time tasks by the fact that they are based on the specification of deadlines by the applications developers.

Since real time scheduling capabilities are required in non real time-only systems, various projects in literature tried to offer integrated support for both real time and best effort tasks. One of the simplest approaches is provided by Linux (see Section 3.3.1), which implements a very simple real time scheduling class at maximum priority. A more articulated approach is provided by ad-hoc process schedulers specifically designed for supporting both real time and regular tasks (two examples are covered in Section 3.3.2) and there exist even whole operating systems targeted for real time (Section 3.3.3).

3.3.1 Vanilla Linux

Real time handling in Linux (referring to the 2.6 vanilla kernel series - where the term '*vanilla*' indicates the main Linux branch managed by Linus Torvalds [73]) is implemented in a quite simple way by giving to real time tasks maximum priority over any other runnable task. So, any real time task that becomes runnable will preempt any other non real time task. More precisely, there exist two real time scheduling classes called `SCHED_FIFO` and `SCHED_RR`, which are allowed to dispose of 95% of the available processor(s) bandwidth. The remaining 5% is reserved for regular tasks to prevent starvation with high real time load. The two classes differ in how they manage the real time tasks [52]:

- `SCHED_FIFO` implements a simple First Come First Served scheduling algorithm, which is not preemptive;
- `SCHED_RR` works according to the Round Robin scheduling algorithm, which is preemptive and based on timeslices.

This approach towards real time scheduling is quite naïve and, in fact, treats real time tasks in a best effort manner, since it does not even allow specifying deadlines. Linux, being targeted at desktop and server systems rather than real time systems, trades off real time accuracy for simplicity.

3.3.2 Integrated Real Time and Best Effort Schedulers

In literature, there are some projects which try to build an integrated process scheduler able to manage both real time and non real time tasks. The motivation for this effort is that, in modern computing systems, the traditional notions of real time and best effort scheduling environment have fractured into a blurred spectrum of classes [16] and thus there is a need for a process scheduler able to handle all these heterogeneous requirements.

Brandt et al. [16] present the Rate-Based Earliest Deadline (RBED): an integrated multi class real time scheduler. RBED provides a unified scheduling approach and it is based on a general model of real time scheduling called Resource Allocation/Dispatching (RAD). The idea at the base of this model is to separate the processes of deciding the share of resources to be granted to a task (i.e., *resource allocation*) and of determining the timing of the delivery of the assigned resources (i.e., *dispatching*). To simultaneously handle real time and non real time tasks, RBED assigns a *target rate of progress* and a *period* to each task; for real time tasks, these parameters are determined based on the deadlines, while for best effort tasks a semi-arbitrary period is assigned with the goal of ensuring high responsiveness. The scheduler, then, dynamically adjusts the periods trying to both meet real time deadlines and not starving any best effort task. RBED has been implemented in Linux 2.4.20 (completely substituting the original scheduler) and experimental evaluations show that it is able to manage tasks from different classes at the price of adding a small scheduling overhead with respect to the default Linux scheduler.

The practice, adopted in RBED [16], of assigning an estimated period to non real time tasks introduces an artificial constraint (the arbitrary period) to tasks which are not characterized in this manner by the application developers [54]. Another interesting work focused on providing an integrated process scheduler with support for best effort along with real time tasks has been proposed by Nieh and Lam [55]. The authors present a Scheduler for Multimedia And Real Time applications (SMART) which, similarly to RBED, is based on the separation of *importance* and *urgency* of tasks:

- the importance of a task is related to its priority assigned from the user at runtime;
- urgency is defined only for real time tasks and it is related to the time constraints expressed by the deadlines.

To measure the importance of a task, SMART uses a *value-tuple* composed of the priority and the Biased Virtual Finishing Time (BVFT) of the task; the priority is a static quantity, while the BVFT is a dynamic quantity used to measure the degree to which each task has been allotted its share of resources. By using the importance and urgency of tasks, SMART does not need (as opposed to RBED) to add pseudo real time constraints to best effort tasks. The authors implemented SMART in the Solaris UNIX operating system [55] and demonstrated superior performance with respect to the UNIX SRV4 scheduler in supporting multimedia applications.

3.3.3 Real Time Operating Systems

A different approach to managing the scheduling of real time tasks along with non real time tasks is the design of a specific operating system, usually referred to as a Real Time Operating System (RTOS). This solution is sometimes applied in industrial contexts with the goal of using commodity hardware instead of expensive and complex ad-hoc controllers [30] to manage real time operations. Some notable examples of RTOSes have been designed for embedded systems, which are often used for mobile devices or industrial control applications, for instance Windows CE, VxWorks, Jbed and others [12]. These OSes are designed for tiny systems with limited resources and do not scale well to more powerful platforms. Other real time operating systems are designed to scale to different platforms and, among these, two notable examples (Real Time Linux (RTLinux) [83] and Real Time Application Interface (RTAI) [59]) allow the coexistence of real time and not real time tasks by directly managing real time tasks and relying on the Linux kernel for non real time operations.

RTLinux [83] enhances the Linux kernel with hard real time capabilities while still supporting all of the features provided by Linux. This is done by creating a real time kernel which directly handles the real time tasks and relies on Linux to do all the non real time management. The real time kernel is kept as simple and deterministic as possible and it intercepts all of the interrupts. If an interrupt refers to a real time task, then the interrupt handler is directly provided by the real time kernel, which offers hard real time guarantees on execution times, providing direct access to the raw hardware. The Linux kernel acts as the idle task of the real time handler and thus it is a fully preemptible process that is executed only when no real time task requires the processor. This structure enforces maximum priority to real

time events, which can preempt the Linux kernel at any time, while still providing all of the capabilities offered by Linux for non real time tasks.

A similar approach is proposed by RTAI [59], which patches the Linux kernel by installing a generic Real Time Hardware Abstraction Layer (RTHAL) [30], which is implemented in a kernel module (i.e. a dynamically loadable extension) which provides real time management. The RTHAL has exclusive direct access to the hardware and it provides a software emulation that lets Linux keep working unchanged for non real time tasks. Real time tasks are managed by the RTHAL module, which supports three scheduling policies [30] (see Section 1.4.6 for definitions):

- a fully preemptible FCFS policy for voluntary cooperative scheduling;
- a Round Robin policy;
- an EDF policy which allows the definition of deadlines.

Thanks to this design, RTAI enables general purpose CPUs to be used in time critical systems such as controllers for aeroservoelastic systems [30].

A common characteristic of RTOSes is presenting a layered structure, with a prioritized layer in charge of dealing with real time tasks and a preemptible layer which manages best effort tasks. This fact makes this kind of systems more suitable for specific applications mainly focused on real time tasks where best effort tasks are a minority. A system where real time and best effort tasks are present in similar amounts, would probably better managed by an integrated scheduler such as the ones illustrated in Section 3.3.2.

3.4 Adaptive Process Schedulers

Recently, along with the ideas on autonomic computing, some research projects in literature propose adaptive process schedulers, i.e. schedulers that have the capability of adapting the scheduling policy in use according to the system status with respect to some specified goals. Adaptive features have been proposed with different goals, from improving cache locality or decreasing lock contention to maximizing applications performance. This Section illustrates some of these works that employ adaptive techniques for process scheduling.

3.4.1 System Status Aware Scheduling

In general, the goal of producing an adaptive scheduler is to get better scheduling decisions by taking in consideration the status of the system and/or the needs of the users. Some adaptive optimization do not require input from the users: two examples are lock contention [82] and cache memory locality [20].

Xian, Srisa-an, and Jiang [82] propose a contention-aware scheduler designed for large Java applications in multicore systems. In this work, they propose a Linux-based system scheduler able to exploit locking information coming from the Virtual Machine (VM) in order to proactively reduce the possibility of lock contention. This is done by taking two actions:

- Dynamically clustering threads that share similar lock-protected structures and serializing each cluster.
- Giving longer execution quanta and higher priority to threads that are in the middle of critical sections.

This approach is shown to achieve up to a 15% performance boost (in terms of execution time) when running large multithreaded applications, at the price of a 3% performance degradation for smaller applications.

Chen et al. [20] work in the opposite context of embedded real time systems and propose a scheduler aware of cache memory usage by the tasks in execution. The aim of this work is to improve the performance by adapting the scheduling decisions to maximize the reuse of cached data. This is done focusing on an embedded execution environment that continuously executes a single application and applying both optimizations at compilation time and adaptive decisions at runtime. Within this very constrained scenario (and with some hypotheses on the application behavior), the authors are able to improve the performance of about 13% over classic real time scheduling algorithms.

3.4.2 Tasks Classification

Other adaptive schedulers try to take into account users' preferences; an example is a work by Lim and Cho [49], who propose an approach to process scheduling based on fuzzy inference, where the idea is to classify the tasks in execution according to their behavior and determine

the preference of the users for each class. With this information, the scheduler should be able to adapt its decisions according to the nature of the running tasks (batch, interactive or real time) and to the user's preference for each class. The authors propose a Linux-based system, where the needed information is gathered with two new modules: a *process classification* module analyzes the tasks logs to classify their behavior and a *user modeling* subsystem analyzes the users' preferences. These preferences are gathered by requiring feedback from the users, which is done by asking some questions when they log out about their feeling on the system performance. All the information is represented by using fuzzy sets and a fuzzy inference engine is then in charge of setting the tasks priority based on this knowledge base. The authors show that the fuzzy rules are able to make the scheduler adapt to the user's expressed preference, but these techniques incur in a relevant overhead with respect to standard scheduling techniques.

3.4.3 Adaptivity for Applications Performance

Adaptive approaches to process scheduling have been applied also to improve the performance, either considering the QoS of multimedia applications or seeking maximum throughput in parallel applications.

The management of multimedia applications is somewhat in the middle between real time and best effort scheduling, as these applications do have QoS requirements, but are often not treated like real time applications (i.e., application programmers do not explicitly insert deadlines in the code). A possible approach to this problem is instrumenting the applications with a suitable API for specifying deadlines and then using a scheduler based on a feedback loop to adapt the amount of allotted CPU time according to the requests and the available resources. This approach, which is similar to the solution employed in this thesis, has been proposed by Cucinotta et al. [26]. This work, however, does not currently support multi threaded applications and uses real time concepts in contrast with the general monitoring approach based on autonomic computing proposed by this thesis. The main focus of the project is on the scheduling policy, which employs control theory techniques.

Another work focusing on applications performance has been proposed by Corbalan, Martorell, and Labarta [21], who deal with the problem of maximizing the performance of parallel applications on multicore systems. The scheduler uses a helper library, called *SelfAnalyzer*,

to retrieve the speedup achieved by an application when it is assigned more or less cores and determines at runtime the optimal number of cores to maximize the application's performance.

3.5 Summary

Monitoring infrastructures in computing systems have been largely used in different contexts and at different levels. Autonomic computing system, however, still lack a monitoring infrastructure able to prove really efficient, scalable, lightweight and easy to use for applications developers and users in a goal oriented environment.

The results shown by existing adaptive schedulers indicate that this component can highly benefit from an autonomic behavior in many different cases and these works show how important gathering meaningful information (through appropriate monitoring infrastructures) is in order to make the autonomic approach really effective. The proposed methodology to enhance a commodity operating system with autonomic capabilities through a generic monitoring infrastructure and the use of this methodology for creating a performance-aware process scheduler on top of Linux are presented in the next Chapter, illustrating in detail the contributions of this thesis.

Chapter 4

Proposed Approach

The proposal of this thesis for the creation of autonomic computing systems is presented in this Chapter, from the underlying vision to the details regarding the proposed methodology to extend a commodity operating system with an autonomic layer. The general approach is illustrated in Section 4.1, where the goals, model and components at the base of the project are defined. The focus is then moved, in Section 4.2, to the direct contributions of this thesis to the creation of an autonomic layer in a commodity OS by providing a monitoring facility and a proof of concept adaptation policy managing performance-aware adaptive process scheduling.

4.1 Vision and High-Level Structure

The work proposed in this thesis embraces the ideas expressed by the autonomic computing community [41] and tries to decline them into a feasible approach to the realization of that vision. The motivation for this effort comes from the observation, shared by different researchers - see the related works illustrated in Chapter 3, that the organization of current computing systems is exposing too much complexity to the software developers. The lack of strong system support for the evolutions seen in computer architectures (e.g., multi- and many-cores and heterogeneous computing units) obliges applications developers to take into account lower-level details about the target architectures, making the software design process more complex. This issue can be addressed with a neat distinction between system and applications developers, where the former are in charge of supporting the computing architectures and offering suitable high level interfaces to the latter.

Within this context, the overall research objective is the creation of self-aware computing systems to tackle the complexity problematics illustrated in Chapter 1, dealing with any kind of modern computing device: from mobile devices and desktops to servers, mainframes and huge computing facilities. This vision is shared with the Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) research group [10], which is founded on the belief that system developers should employ techniques based on the ideas of autonomic computing for enabling computing systems to continuously adapt towards optimal performance. Within this context, the concept of performance is extended beyond the mere idea that "*the faster, the better*", but it comes to include objectives such as minimization of power consumption and thermal efficiency together with the goal of ensuring to the users an experience as close as possible to their needs.

The proposed approach to realize this vision in actual computing systems is based on the idea of leveraging the ODA control loop (see Section 1.2.2) at different levels within the system. Starting from the architectural level, each component can benefit from internal ODA loops to realize autonomic management "in the small"; proceeding at a higher level, broader ODA loops should orchestrate the different subsystems and, at the top level, a system-wide control loop, aware of the system as a whole, should be in charge of pursuing maximum runtime performance (in the broad sense explained above).

4.1.1 Goals and Contributions

The long term goal of the CHANGE group [10] is the realization of the autonomic revolution introduced in Chapter 1 by creating methodologies and designs for computing systems able to adapt their behavior according to their internal and environmental status and to optimize the running applications in order to ensure a consistent user experience on many different architectures and in different environments. To do so, the group works on various aspects of computing systems, from architectures to operating systems and development tools. The aim is to allow application developers to concentrate on what their applications must do, leaving all the architecture-dependent details to be managed by the autonomic features of the systems where they will be deployed. To come to this scenario, all of the components of a computing system could be modified in order to create an autonomic behavior in the system as a whole. Within this context, the first and most important system layer to be reworked in

an autonomic direction is the operating system; this is true for at least three reasons:

- The OS is the system layer which exposes the system resources towards the applications; hence, it has a direct link with the applications, which are the entities that the autonomic system must serve according to their performance requirements.
- The operating system has, on the other side, direct access to the hardware resources and it is in charge of managing them.
- Since the OS is a software system, it is possible to work at this level in an agile way, without the need of requiring hardware modifications to the architectures or to the components. This could be a further step to improve the autonomic features once the autonomic base system in the OS layer will be ready.

To sum up, the OS is the glue between the hardware and the applications and it is possible to work with relative ease at this level. Therefore, the first goal on the road to autonomic computing is the realization of an autonomic layer within the OS: this layer will serve as the basis and support for successive improvements both at architectural and applicative level. Within this context, the contributions of this thesis are:

- The description and formalization of the high-level approach towards autonomic computing (which has been defined in collaboration with the CHANGE research group [10]); this matter is covered in Sections 4.1.2 and 4.1.3.
- The proposal of a methodology for creating operating system-level autonomic capabilities leveraging the ODA control loop (refer to Section 4.2) and, in details:
  - the design of a general purpose monitoring infrastructure to provide the observation phase;
  - the demonstration of the applicability of the proposed approach by employing the monitor for performance-aware adaptive process scheduling.
- The implementation of both the monitor and the adaptive process scheduler over the Linux kernel [73] (this is treated in Chapter 5).
- The evaluation and characterization of the implemented system with both ad-hoc micro benchmarks and real workloads (experimental results are presented in Chapter 6).

The remainder of this thesis gives a detailed report of these contributions.

4.1.2 Autonomic Computing System Model

Most modern computing systems can be subdivided into three layers: *Hardware Components*, *operating system* and *Applications* (see Section 1.2.3); the model for an autonomic computing system should not distort this well established structure, but augment it:

- Within the hardware layer, each component should embed integrated autonomic controllers to autonomously manage its lower level parameters in order to maintain a stable working status.
- At the upper level, the applications should embed similar software mechanisms to tune their behavior, taking into consideration also the users' preferences.
- Both the hardware components and the applications should expose information about their status towards the OS which should use these data to implement a number of ODA loops using a combination of the available information sources to take decisions and act onto the OS subsystems, which in turn will alter the runtime status.

As argued in 4.1.1, the first place where to act in extending the classic computing system model is the last in the above list, i.e., the operating system; this vision is represented in 4.1, which shows some examples of interaction between the system components, the autonomic layer and the applications. The Figure represents an extension of the classic three-layered structure of a computing system, where an autonomic layer has been added within the operating system. This autonomic layer contains a number of autonomic controllers using information coming from the hardware architecture and the software in execution in order to optimize the runtime status; more in details:

- The applications are allowed to explicitly communicate to the autonomic layer their performance goals (which can be specified by the developers or the users) and are continuously monitored to capture any deviation from the required performance.
- The hardware components are monitored both in terms of performance and in terms of health status (for instance, working temperature, voltage, . . .) and their performance

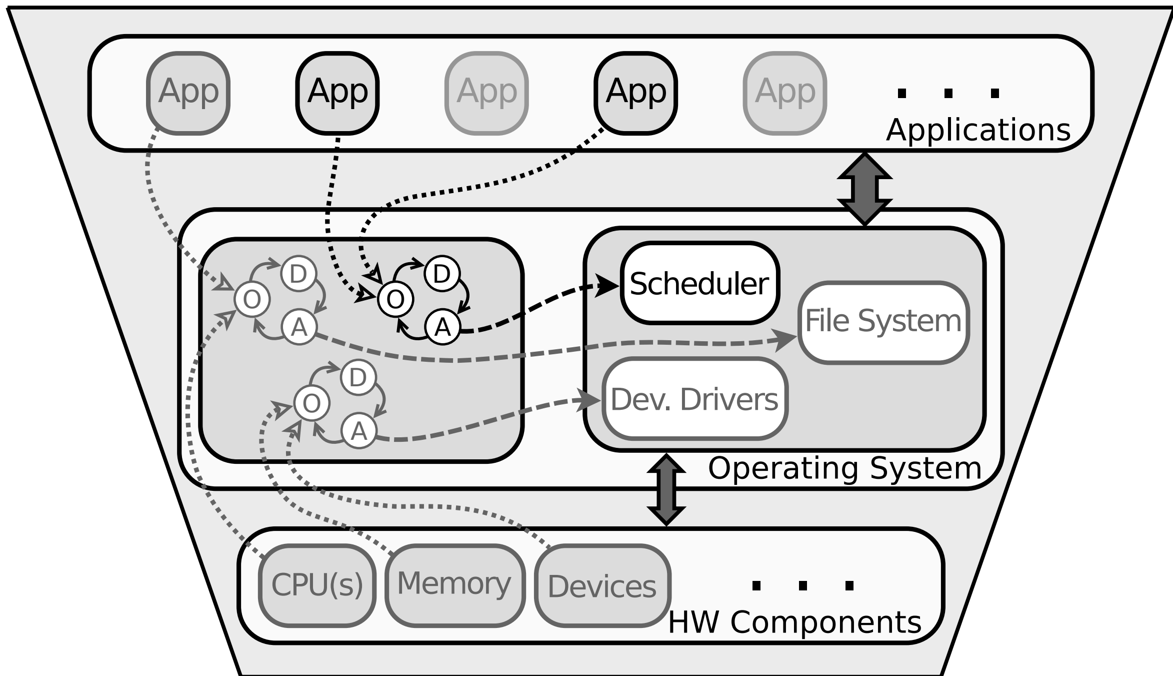


Figure 4.1: Proposed model for an autonomic computing system

capabilities and health restrictions (such as maximum working temperature) are considered with respect to the current conditions.

- The components of the operating system are controlled (according to a certain policy) in order to apply any needed modifications aimed at making the system fit into the status space defined by the goals expressed by the applications and the runtime system constraints. These components can then affect both the applications or the hardware components, thus closing the control loop.

For instance, the ODA loop highlighted in the Figure gets information (i.e., status and goals) from the running applications through a monitoring facility and is able to compare the status with the goals, determining how to act on the process scheduler. This process has been completely implemented for this thesis as a proof of concept to illustrate the capabilities of this enabling technology.

4.1.3 Autonomic Components

The *observation*, *decision* and *action* phases in the ODA loops present within the autonomic computing system are realized through two principal *autonomic components*:

- *monitors* preside over the observation phase;
- *adaptation policies* manage the decision and action phases.

More in detail, a monitor provides an interface between the system and users on one side and the autonomic layer on the other: it fetches relevant information from the system and goals from the users and it makes these data available to the other autonomic components offering a suitable API. A monitor is characterized by what measure it records (called the *target* measure) and it can span different levels of the system. For instance, there could be a temperature monitor that simply records and makes available to the autonomic layer the thermal state of part of the system, thus bringing information from the hardware to the OS layer; another kind of monitor can measure the performance of the running applications according to some appropriate metric. A monitor must also provide a means of specifying which are the desired values for its target measure. For instance, a performance monitor must provide a way of stating performance goals for the monitored applications in terms of a range of desired values *in the metric* used by the monitor that represent the desired runtime state for each application. This range of desired values is the *goal* for that target measure.

Adaptation policies access the information provided by one or more monitors and are able to compare the measures with the goals, using a decision mechanism (e.g., based on machine learning or control theory) to determine whether any corrective action is needed. If the measures do not match the goals, an adaptation policy disposes of one or more *actuation hooks* within the system (e.g. within a device driver or the process scheduler) where it can modify some parameters (e.g. the clock frequency of a processor or the CPU time assigned to an application) to alter the system status. In this way, each adaptation policy, coupled with one or more monitors, identifies a separate ODA loop within the autonomic system.

Since within the same system different adaptation policies may coexist and have contrasting goals or clash in the use of actuation hooks, there is the need for an higher level component in charge of coordinating the operation of the adaptation policies. This component, named *adaptation manager*, has the role of coordinating the autonomic action and has access to all

the monitors and has visibility towards the system as a whole. A representation of a snapshot of a possible status of the autonomic layer of a computing system is represented in Figure 4.2. The Figure represents three monitors gathering measures and goals for three different targets

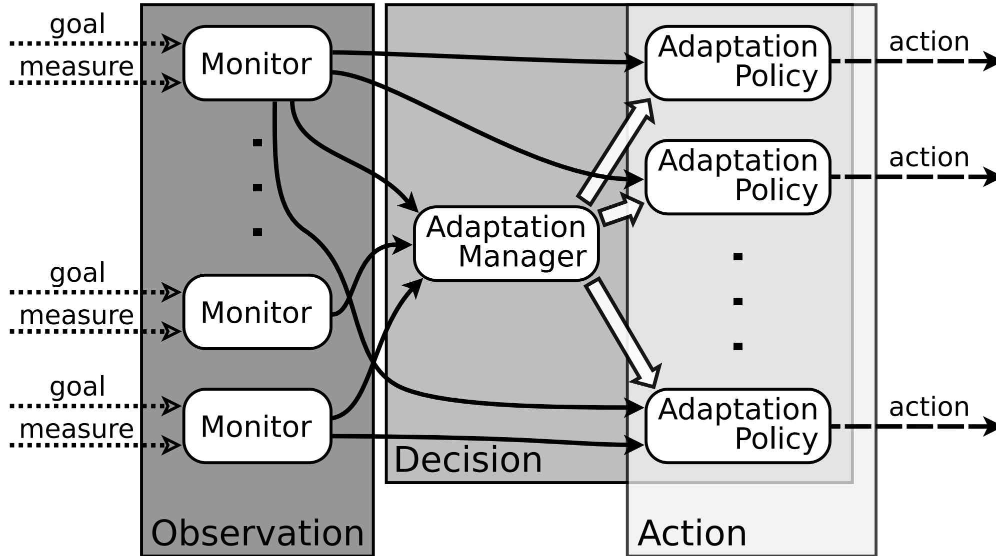


Figure 4.2: Autonomic components operating within the autonomic computing system model

and three adaptation policy getting monitoring information from one or more of the available monitors and acting on the system as needed. The Figure also shows the adaptation manager, which has view over all the monitors and can act on the adaptation policies enabling or disabling their operation with the aim to reach the global system goals. For instance, there could be different adaptation policies working on the same target and using the same monitoring information but different decision mechanisms (e.g., an heuristic versus a control theory-based policy); in this case, the adaptation manager would be in charge of choosing the best policy according to the runtime context.

The work proposed in this thesis is focused on the development, at the operating system level, of a software monitor, which provides a way of getting information from the running applications and an adaptation policy using this information to act on the process scheduling process. These two components create a first OS-level ODA autonomic control loop and build a base for the future realization of more monitors and adaptation policies. Since, up to now, only one ODA loop exists, there is no need for the adaptation manager, which is left for future work.

4.2 AcOS: an Autonomic Operating System

The structure proposed in the first part of this Chapter describes the general approach towards the creation of autonomic management capabilities in computing systems. As already argued, the first level where to work in this direction is the operating system, where an autonomic layer must be created. One of the design choices made at the beginning of this work was whether starting to build a new operating system designed for being autonomic or modifying an existing commodity operating system and enhance it. Creating a new operating system would grant great flexibility, but would impose an enormous overhead in building all the low-level structure for interfacing towards the hardware resources, requiring a great amount of work to be done prior to being able to focus onto the autonomic features. On the other hand, extending an existing operating system allows to reuse what already exists, even though reducing the design flexibility. The chosen approach is extending the Linux kernel to create a base for an Autonomic OS (AcOS) with a first autonomic control loop, which can be incrementally extended in future works realizing more monitors, adaptation policies and the adaptation manager. The use of Linux as the base for AcOS offers some advantages with respect to extending a different operating system or starting the development of the new OS from scratch:

- Linux is a modern OS with support for different architectures and its source code is freely accessible and modifiable (which is not true for some major commodity OSes, e.g., Microsoft Windows or Mac OSX).
- Linux is widespread and an autonomic framework on top of it allows to keep full compatibility with legacy applications and to offer a well known development environment for the creation of new applications.
- The open and community-based development style of Linux allows to directly access all the source code and easily find documentation and support. Moreover, Linux is continuously tested against security bugs and any fix distributed for Linux is automatically available for the Linux-based autonomic OS.

The extension of Linux proposed in this thesis, serving as a base for AcOS, consists in a software monitor based on the abstraction of *heartbeats* (the same used in Application Heartbeats

- see Section 3.1.3) and an adaptation policy exploiting the monitor to measure applications performance for enhancing the default process scheduler in Linux (which is the CFS - see Section 3.2.2).

The choice of using Linux as the base for AcOS drove the design process towards placing both the monitor and the adaptation policy in kernelspace. This choice is the most natural in a monolithic kernel such as Linux, which implements the process scheduler fully in kernelspace. Moreover, the proposed monitor is designed to offer improved functionality and performance with respect to AH, which is completely implemented in userspace [37] and, even though being open source software, could not represent a solid alternative to realize the goals of this project. The remaining of this Section offers a thorough illustration of the design choices at the base of the different components of the proposed system.

4.2.1 Monitoring Applications' Performance

In order to create an autonomic control loop able to realize adaptive scheduling with the aim of helping the applications in execution reach their performance goals, the first step is to clarify how applications' *performance* can be measured in a simple and efficient way. To do so, it is important to remember the definition of *task* in Linux (see Definition 1.7), which is the atomic schedulable entity. Some other definitions are useful to understand the monitoring framework that has been defined; first of all, Definition 4.1 precises the meaning of the term *application*.

Definition 4.1 (Application)**.** *An* application *is any program in execution; it can be a single process, a thread group or a group of cooperating processes.*

In general, the applications are the entities to be monitored and thus monitoring an application can involve the measurement of the performance of a single process, a group of threads or a group of cooperating processes. In general, the performance monitoring of an application involves dealing with one or more tasks. More in details, an application to be monitored could be structured in different ways in terms of parallelism:

- An application could be a single process, autonomously doing its job.
- An application could exploit thread level parallelism by letting a set of threads work in parallel.

- More than one process could cooperate by the means of IPC in an application exploiting process level parallelism.

All these types of applications must be correctly managed by the adaptive scheduling ODA control loop; for this purpose, it is useful to define the concept of *group*, which is explained in Definition 4.2.

Definition 4.2 (Group)**.** *A* group *is an ensemble of tasks which share performance monitoring.*

The concept of *group* is intended to allow the monitoring of applications that exploit parallelism in various ways. For this reason, a group can be composed in three different ways:

- it can include a single task;
- it can be *intra-application* (the members are threads with a common parent);
- it can be *inter-application* (the members are tasks belonging to different processes).

The first possibility is for monitoring single-threaded applications, the second allows monitoring applications that use thread-level parallelism and the last can be used to monitor applications that are based on process-level parallelism (the lack of support for process-level parallelism is one of the shortcomings of Application Heartbeats). The concept of group is important because it really defines the performance-monitored atom. In fact, a group is used to aggregate the performance monitoring of different tasks that cooperate to perform a common job: measuring the performance of each task alone does not make much sense, but the performance of the group is what counts. For this reason, each group must be characterized by a current performance indicator and a performance goal; the question is now how to represent them.

Probably, the most direct way of fetching first-hand information about the performance of a group of tasks is asking the task themselves to signal how they are proceeding. To do so, however, a simple way to instrument the software is required, which must be easy to use for applications developers. The proposed monitoring infrastructure is based on the idea of making the tasks of a monitored group emit *heartbeats*; this concept is clarified in Definition 4.3).

Definition 4.3 (Heartbeat)**.** *A* heartbeat *is an atomic signal sent from an group (i.e. by one of its component tasks).*

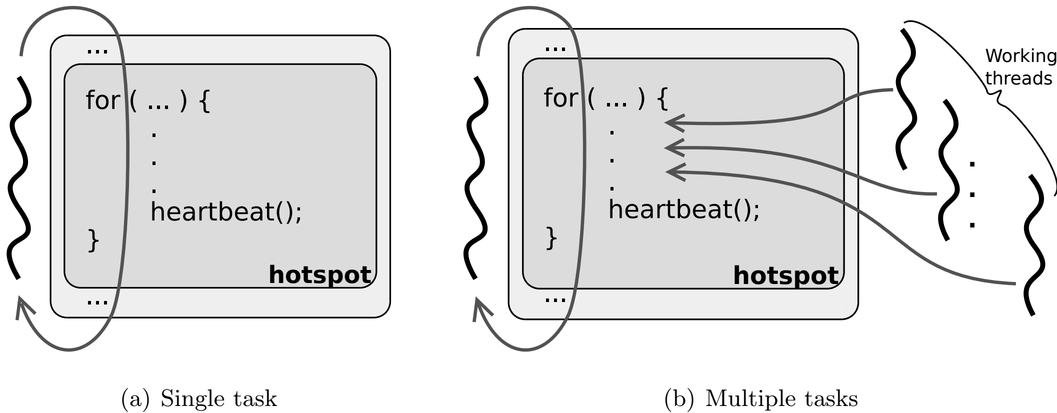
The concept of heartbeats has been used in literature mainly for monitoring availability [71] and, depending on how heartbeats are used to instrument a group, it is possible to exploit them to monitor different properties (for instance, for monitoring lock contention [31]). Within this work, heartbeats are used in a similar way to how they are employed by Application Heartbeats [37]: an heartbeat must be emitted by a group to indicate progress in its execution.

To be suitable for being monitored in performance by issuing heartbeats, a group must be characterized by at least one CPU-bound section of code referred to as a *hotspot* (see Definition 4.4), which is where heartbeats are emitted.

Definition 4.4 (Hotspot)**.** *A* hotspot *is a performance-relevant code section executed by an application.*

Within each group, there will be at most one active hotspot at the same time, which may be structured in different ways: it may belong to a single task, be executed in parallel by several tasks or be spread into parts, each executed by a single task that does some independent work, and then converging to a single point. In general, the hotspot coincides with a loop into which a repetitive work is performed (either by a single task or by several tasks in parallel) and the end of the hotspot is usually where the application issues heartbeats to indicate that an iteration has been completed. The hotspot is owned by the tasks which execute its code and each task accounts the number of heartbeats it emitted while executing that code. Two examples of hotspot are represented in Figure 4.3. Figure 4.3(a) represents an application where a single task does all the work throughout the hotspot and, at every iteration, it emits an heartbeat. In this case, there will be a unique group containing the task. Figure 4.3(b) shows a scenario in which a main task controls the work (with some adequate synchronization mechanism), while the body of the loop is in fact executed by some other tasks, which may work in parallel. In this case, the heartbeat may be emitted by the controlling task, after every thread has finished its part of the job; in this situation, there would be again only one group formed by the main task alone. Another possibility is that the tasks performing the work (for instance, in the case of a thread pool) are be grouped together and emit an heartbeat each at the end of their job.

Another useful concept regarding the state of a task is its *activity*, which is defined in



(a) Single task (b) Multiple tasks

Figure 4.3: Graphical representation of an hotspot

Definition 4.5.

Definition 4.5 (Task activity)**.** *A monitored task is in an* active *state if it is currently executing a code section where heartbeats are emitted; i.e. if it is executing the code of an a hotspot.*

The activity of a task should be taken into consideration by the actuation policies to decide whether or not to act on the task. For instance, in the situation proposed above, where there is a "father" task that spawns some "son" tasks that actually perform the work and emit heartbeats, the father could initially create the group and remain inactive, while the sons add themselves to the group and become active when they begin executing the cyclical code. Within this context, the father is simply waiting for the sons to do the work and thus it must not be affected by actions based on the group performance; this information is exposed through the activity state of the tasks.

According to the description proposed up to now, an application may be organized in groups, each of which can contain one or more tasks. Each group, to be interesting from a performance-monitoring point of view, contains a hotspot and its tasks emit a heartbeat each time they terminate the execution of the hotspot. Each task, then, owns a counter (similar to the event counters found in monitoring literature - see 3.1) showing the total number of heartbeats it emitted while executing the hotspot. Based on this setup, it seems logical to used the metric defined in Definition 4.6 to measure the current performance of a group.

Definition 4.6 (Heart rate)**.** *The heart rate of a group is the number of heartbeats issued per time unit by its tasks; it is measured in* $\frac{heartbeats}{second}$.

The definition of heart rate allows to answer to the question regarding how to represent the current performance indicator and the performance goal. Each group is characterized by a current heart rate and a goal heart rate. The current heart may be evaluated over the whole execution time of a hotstpot (in this case it is referred to as *global heart rate*), or over a certain time window (in which case it becomes a *window heart rate*). Intuitively, the faster a hotspot is executed, at a higher rate the heartbeat counters of the tasks will increase (i.e., there will be a higher throughput), yielding a higher heart rate. Hence, the current heart rate is suitable to be used as an indicator for the current performance of the group. The performance goal, on the other hand, is represented as a goal heart rate (a broader discussion on this matter may be found in Section 4.2.5).

The monitoring infrastructure designed and developed in this thesis is based on the concepts just exposed. The monitor works by receiving the heartbeats emitted by the tasks in the monitored groups and efficiently yielding statistics in the form of heart rates. In fact, such metric is not specific to performance monitoring and, depending on how the applications are instrumented, it provides a quite generic infrastructure for different measurements. As outlined above, this infrastructure is suitable to be used for performance monitoring of applications characterized by one or more CPU-bound sections which are repeatedly executed and represented as hotspots. The periodic behavior of the hotspot guarantees that the heart rates reported by the monitor are meaningful both as a description of the past performance of the group and as an estimation of its future behavior if the border conditions are maintained. This property is very important, since the monitor is used in a ODA closed loop controller. Clearly, this is a limitation to the scope of applications suitable to be monitored, but it is necessary to ensure that the data gathered by the monitor are meaningful, which is crucial for the control loop to work as expected.

4.2.2 Heart Rate and Real Time

Applications with a periodic behavior are often taken into account in real time contexts, for instance by Cucinotta et al. [26] (see Section 3.4.3). The hypothesis of periodicity, when dealing with real time, allows to simplify the model by considering each task as split into

a sequence of jobs characterized by the same worst-case execution time. Then, a deadline is associated to each job, which must absolutely terminate its execution before the deadline (in case of hard real time) or, in the case of soft real time, must ensure a certain QoS (i.e. do not miss the deadline on average, with bounded miss rate).

While the deadlines used in a real time context are set *locally* for each iteration of the periodic job, the heart rate of a hotspot provides a measure *on average* of the speed at which a hotspot is being executed (i.e., a measure of its average throughput). The heart rate is an aggregated measure which gives a global view of the average execution rate, but does not consider deadlines on each iteration. For this reason, using a heart rate to characterize the performance of an application is very different from a hard real time approach, to which the concept of specific deadlines on each job is crucial.

The heart rate, as used in this thesis, can be used to represent the current QoS of a group; this is similar to what is done is soft real time systems. In fact, the goal (defined as a desired heart rate) is a requirement of average execution behavior with respect to the current heart rate i.e., it defines a certain desired quality of service in the execution of a hotspot. This is similar to a soft real time system where a periodic task is requested, at each iteration (i.e., for each job), to match a deadline on average. The heart rate, however, brings no knowledge of how fast each iteration was executed and it trades off some precision for more simplicity. For this reason, a heart rate is more suitable in a *best effort* context, where no guarantees are given, but it is possible to try and satisfy at the best of the system's capabilities, the desired QoS of the monitored applications.

4.2.3 The Heart Rate Monitor

The monitoring facility created for this thesis is based on the concepts illustrated in Section 4.2.1 and its function is to expose information about the heart rates of the groups of tasks, which constitute the monitored applications; hence, the monitor has been called Heart Rate Monitor (HRM). As already argued, this approach is not limited to performance monitoring and, depending on how the code is instrumented, the heart rate can represent other properties, such as the contention over a lock. For this reason, HRM has been designed to be as general as possible and its use has then been specialized to performance monitoring; moreover, the monitor has been designed for Linux, but its structure must be easily portable

to any other monolithic kernel and, possibly, also to non monolithic operating systems. With these considerations in mind, the design of HRM has been based on some principles:

- The monitor should be as lightweight as possible on the monitored applications.
- The impact of the monitor on the performance should be zero for non monitored applications.
- The structure of the monitor should limit the need of locking on shared data structures.
- The information gathered by the monitor should be easily accessible by adaptation policies working inside the kernel and the same information should be made available also to adaptation policies operating in userspace.

The first principle is important because the monitor must be able to gather meaningful information on the performance and thus the overhead given by the measurements on the execution of the tasks must be minimal. At the same time, the monitor should have no influence on tasks that are not being monitored, ensuring the same execution performance as in a system where the monitor does not exist. These two principles are matched in the design of the monitor by leveraging the following properties:

- The computation of the statistics (i.e. the heart rates of the groups) is decoupled from the emission of heartbeats by the monitored applications.
- The tasks must register by using a certain API to enable performance monitoring; by using this API, applications developers can instrument their software specifying how the monitored groups are composed.

Decoupling the heartbeats emission and the statistics computation (which is an improvement with respect to the design of the *Application Heartbeats API* used in the SEEC framework[38]) is done by moving the computation of the heart rates in a routine periodically executed in the context of a high resolution timer (high resolution timers are available in most modern kernels, such as Linux and FreeBSD). This feature allows minimal direct impact of the monitor on the monitored tasks (as low as a single atomic increment on a counter to emit a heartbeat), matching the first principle. Moreover, the tasks must register, by the means of a function

call to a provided API, to activate performance monitoring; thus, the direct overhead on non monitored applications is zero.

Within Linux (in contrast with message passing-based environments), shared data structures are commonly used for communication among different execution entities. This mechanism is simple and convenient in many cases, but it can lead to performance issues, particularly in multicore or multiprocessor systems, where a poor design of shared data may generate useless traffic on the buses to maintain cache coherency, thus causing a huge overhead on the system. The proposed design uses one or more memory pages (a page weighs 4KB in Linux on most architectures) per group, which are mapped both into the user address space (allowing access to the tasks of the group) and into the kernel address space (allowing the asynchronous computation of the heart rates). Each task member of a group is assigned a field in the shared memory containing a counter which is to be atomically incremented for each emitter heartbeat. The shared pages also contain the performance statistics for the group, which are computed on the kernel side based on the heartbeats counters. This design allows group-wide visibility of the performance information but limits the cache coherency traffic as each task increments only its own counter and does not access the others (more in details, cache alignment of the different fields in the page has been considered during the implementation phase to avoid incurring in false sharing issues). Moreover, the heartbeats emission is completely lockless and locking is used only for group management (adding and removing tasks) and statistics computation.

Finally, the heart rates of the monitored groups are easily made available to adaptation policies running inside the kernel by means of a global groups list and can be shown to userspace adaptation policies by using a shared page and a registration mechanism similar to the one used for task monitoring registration. Also the goal heart rate of the group is stored within its shared memory area and it may be changed by the application itself or by the user at runtime; the adaptation policies can get this information by simply reading the related fields.

A graphical representation of the proposed design for a group is given in Figure 4.4, where the read/write interactions of the various agents (i.e. tasks, statistics computation agent, adaptation policies and users) are highlighted. As the Figure shows, each task members of the group is assigned a proprietary counter which it can increment (which is an atomic operation)

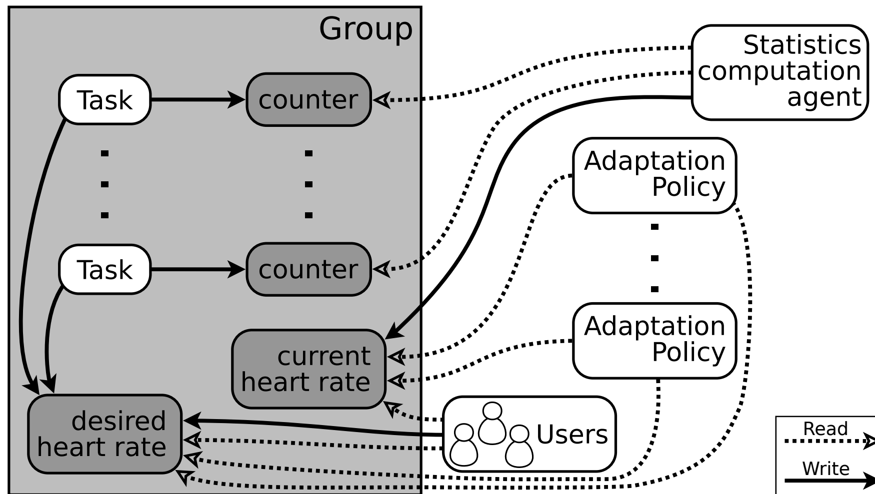


Figure 4.4: Design of a group of performance monitored tasks

to emit an heartbeat. The Statistics computation agent runs asynchronously in the context of a high resolution timer and it writes the current heart rate in a specific field after having computed it based on the values read from the counters. The adaptation policies are allowed to read the current heart rate: this, coupled with the capability of the adaptation policies to read the desired heart rate, is the link between the observation and decision phases for a ODA loop using HRM. Also the users can access to the current heart rate of the group for informational purposes (this is allowed, in the proposed implementation, through the `procfs` pseudo filesystem). The users can also read and adjust the desired heart rate for the group, which can also be modified by the tasks (through calls to the monitor's API). Except for the desired heart rate, which is not modified very often and does not pose synchronization issues, no field in the structure of the group is written by more than one entity; moreover, the increments of the counters are atomic operations and thus the emission of heartbeats is completely lockless. With this design, locks are needed only when computing the statistics (to ensure that no task dies in the meanwhile) and for adding/removing tasks to/from a group. As the analysis presented in Chapter 6 shows, this structure allow a very small overhead of HRM over the monitored applications.

Considering HRM as part of an ODA control loop, it interacts with two different kinds of entities: applications, by registering groups of their tasks and issuing heartbeats, act as *producers*, while the adaptation policies which make use of the monitor acts as *consumers*, as

they "consume" the monitoring information produced by the applications through HRM. The exact form of these monitoring information (i.e., the current heart rate and the goal heart rate) is better defined in the next two Sections.

4.2.4 Provided Statistics

As illustrated above, the performance metric used by the proposed monitor is the heart rate of the monitored groups; this information is computed by a routine which is periodically (with a period of one *time slot*) awaken by a high resolution timer. More in details, two different statistics are computed for each group:

- the global group heart rate;
- the heart rate on a specified window.

The global group heart rate is simply the total sum of the heartbeats emitted by the tasks members of the group divided by the total monitoring time. More precisely, the global heart rate for group g composed by the tasks $t_1, t_2, \ldots, t_N$ which has been monitored for t seconds is given by Equation (4.1).

$$global_hr_g(t) = \frac{\sum_{i=0}^{N} counter_i(t)}{t} \qquad \left[\frac{heartbeats}{second}\right] \tag{4.1}$$

The window heart rate of a group is defined on the last W time slots (which are $w = W \times timeslot_duration$ seconds long) by the formula shown in Equation (4.2), where the meaning of the symbols is the same as in Equation (4.1).

$$window_hr_g(t) = \frac{\sum_{i=0}^{N} counter_i(t) - \sum_{i=0}^{N} counter_i(t-w)}{w} \qquad \left[\frac{heartbeats}{second}\right] \tag{4.2}$$

The global and window heart rates may be used alternatively or contemporary by adaptation policies according to what option fits better their needs. The duration of the time slot and the length of the window are parameters of the system and can be adjusted at run time for each group according to the needs of the monitored application. These parameters are tightly related to the performance goal that developers and users require for an application; this matter is discussed in Section 4.2.5.

The design traits exposed above define the main points upon which the realization of HRM is based. More details on the realized implementation of the monitor are given in Section 5.2.

4.2.5 Desired Heart Rate and Performance Goal

According to the methodology described in Section 4.1.3, monitors act as interfaces between the system and the users and the autonomic layer by providing a measure and a goal regarding a certain target. Following this prescription, HRM provides, beyond the global and window heart rate, a means for applications and users to specify a desired heart rate, serving as a goal for the adaptation policies using the monitor. The most simple way to specify a desired heart rate would be to use a single value. In that case, for instance, an adaptation policy would know that if the current heart rate were smaller than the goal, the monitored group would be too slow. Setting a single value, however, can be limiting because it is often the case that not a single value is admissible, but a range going from a minimum to a maximum. Clearly, the interpretation of such range depends on the adaptation policy which uses the monitor and on how the applications are instrumented.

The choice of using a desired range, between a minimum and a maximum heart rate seemed reasonable for HRM, since it is generic enough to fit different uses for the monitor and (as explained shortly) allows a useful interpretation when HRM is used for monitoring the applications' performance.

In fact, the desired heart rate, despite being specified by the monitor, needs to be consistent with the adaptation policies that make use of it. More precisely, the goal must be consistent with the measure provided by the monitor, but it also must be expressed in a suitable way for the policy to use it for making its decisions. This means that it is necessary to give an interpretation of the desired heart rate range in terms of performance monitoring, which is what HRM is used for by the adaptive scheduling adaptation policy. Considering that the adaptive scheduler is based on a *best effort* paradigm, an deals with trying to match a desired Quality of Service, an interesting interpretation of the desired heart rate range is represented in Figure 4.5. The graph shown in the Figure represents the measured heart rate of a monitored group; the desired heart rate is the lighter shaded area defined by a *minimum heart rate* and a *maximum heart rate* thresholds. Since the heart rate measures the throughput of a group within a hotspot, it can be used to measure if the group is being executed *fast enough* to ensure a certain quality of service. Hence, it is possible to link the QoS with the heart rate in the following way:

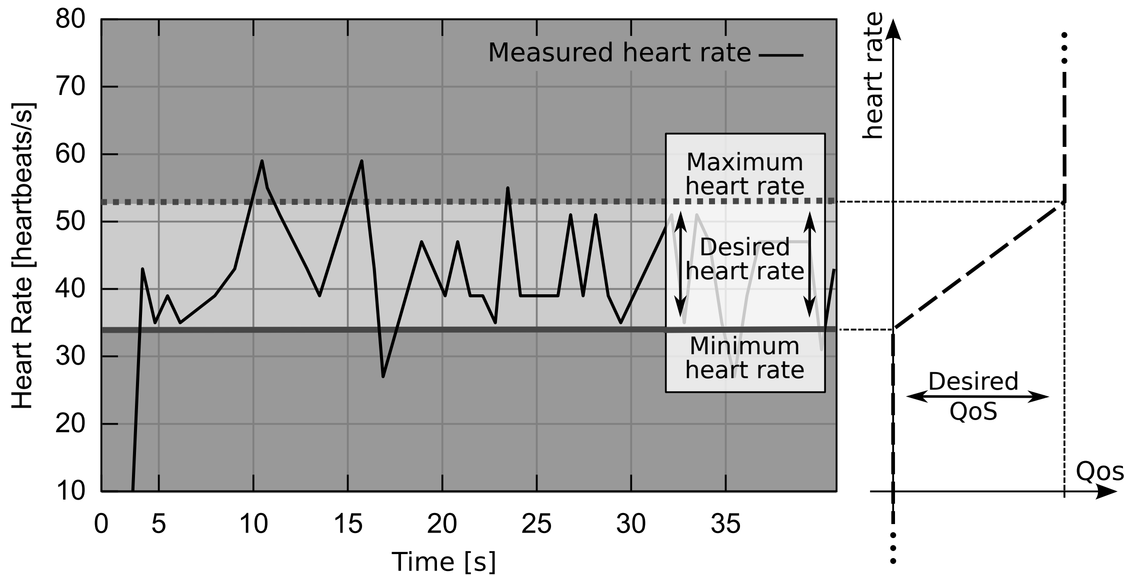


Figure 4.5: Representation of the used performance goal in terms of an interval of desired heart rates between a minimum and a maximum

- The *minimum heart rate* is the lowest heart rate that ensures the required quality of service. AcOS always tries to bring the performance of all the monitored applications over their respective minimum heart rates.
- The *maximum heart rate* is an upper bound to the required quality of service in the sense that further increasing the heart rate would not bring relevant benefits in term of QoS. This value determines a relevant action by the system (i.e. a reduction of the application performance) only if the currently executing applications are determining contention on the system resources. If there are enough resources to let all the applications run over their maximum heart rate, AcOS will not strive to unnecessarily slow down the applications and they will run over the limit[1].

Thus, the minimum heart rate is a *hard* bound, in the sense that the system will always do its best to keep all the applications over this limit, so as to ensure the wanted QoS to each. The maximum heart rate is rather a *soft* bound, which is used mainly to decide which application(s) to slow down when in the need of transferring resources to an application that

[1]Note that there are other possible interpretations of the maximum heart rate bound which define it as a '*hard*' limit, as discussed below.

is running below its minimum heart rate. Of course, in this case, the applications that are penalized are the ones that are currently overperforming their upper bound.

Notice that using the minimum and maximum heart rates as a hard and a soft bound totally depends on the interpretation of the goal heart rate given while using HRM as an application performance monitor within a best effort adaptive scheduling control loop. A different interpretation could be given, for instance by binding the maximum heart rate to power consumption: in this case, the upper bound would be the strong one. This example shows that the desired heart rate range is a flexible way of specifying the goal heart rate, as it admits different interpretations according to the adaptation policy which uses HRM and to what measure it is used for.

Besides the lower and upper bounds on the heart rate, the other parameters that define the performance goal are the duration of the time slot used for sampling the heart rates and the size of the considered window. HRM allows to tune both these parameters at run time for each monitored group. A correct setup of the time slot is important to ensure that the monitoring information (in terms of both global and - if used - window heart rates) is meaningful. For instance, an application with a very variable heart rate should be monitored with a small time slot to be able to catch all the variations. Another problem with the setting of these parameters could incur when monitoring a slow application characterized by a low heart rate with a time slot and a window size too short (i.e., sampling at a too high frequency and using a too small window). In this case, the application would be likely to not have emitted any heartbeat within the window, leading to a misleading measure of zero heart rate on the window.

In sum, the performance goal for a monitored group is defined as a desired range of heart rates (where the lower bound is *hard* and the upper bound is *soft*) which is linked with a requirement in terms of QoS. The duty of an adaptation policy using HRM in this way to monitor the applications' performance is to act on the system so as to make the current heart rate (either global or over a window) stay above the minimum heart rate, stealing, if needed, resources from groups overperforming their upper bound and transferring them to groups running below the lower bound.

4.2.6 The Performance Aware Fair Scheduler

The Performance Aware Fair Scheduler (PAFS) is the adaptation policy that, together with HRM, is used to realize a performance aware adaptive scheduling control loop in AcOS. This adaptation policy tries to balance the assignment of the available computing resources to both monitored (through HRM) and legacy non monitored applications, boosting the possibility for the formers to meet their performance goals and keeping the fairness properties for the latter. It is to be noted that Performance Aware Fair Scheduler (PAFS) is a best effort scheduler and it does not give guarantees on the achieved performance, thus not belonging to the real-time kingdom, but it explores the possibility of coupling a best-effort approach with the ability of still driving the performance towards goals in terms of QoS (as explained in Section 4.2.5).

The main idea underlying PAFS is to alter the process scheduler used in Linux (which is the CFS - presented in Section 3.2.2) in order to make it aware of the monitoring data coming from HRM. The CFS is, as the name says, fair. This means that it strives to give to each concurrently running task the same amount of processor time on average. One of the consequences of this feature (and of how it is managed) is that it is able to ensure non starvation of any task even under heavy workloads; this is achieved while ensuring good responsiveness and without losing time in trying to characterize the tasks' behavior to understand if they require interactivity. In the CFS, the management of the runqueues is done by keeping the tasks ordered in a *red-black tree* (i.e. a type of balanced binary tree), where the key is their *vruntime*. The vruntime of a task is (see Definition 3.1) a quantity representing the ideal runtime of the task as if it were executed on a machine supporting perfect parallelism (i.e. with unbounded computing resources). At each scheduling *tick* (or whenever a context switch is required), the task with the lowest runtime (i.e. the leftmost in the tree) is the one that is picked up for execution. The vruntime of the currently running task is periodically updated by taking in consideration the execution context, which includes how long the task was actually allowed to execute, the number of tasks on the runqueue, and their priorities (i.e., their `nice` value) [52]. As the vruntimes increase, the runqueue is kept ordered and the task with the lowest vruntime is always chosen. These properties of the CFS make it an interesting scheduling algorithm; the CFS does not have any means of knowing how the applications it schedules are performing with respect to the expectations of the developers or users. This

scheduling algorithm, despite considering to some extent the execution context, is based on the only aim of achieving fairness, which sometimes is not the best way to go for yielding a good QoS. PAFS has been created to overcome this limitation, keeping the good properties of the CFS but altering how it treats HRM-enabled applications, which offer their current and goal heart rate as additional information to be used by the scheduler. In brief, the idea underlying PAFS is to modify the vruntime update mechanism for the monitored applications, using the monitoring information provided by HRM to compute its increment. By doing so, the scheduler becomes aware of the performance of the applications for which the additional information is available, while it keeps being fail towards the legacy tasks.

Suppose that the task τ_i is currently in execution on a certain processor (i.e., it currently has the lowest vruntime among the tasks on that runqueue); if at time t either a context switch or a scheduler tick occurs, the vruntime of the task v_{τ_i} is updated according the Equation (4.3).

$$v_{\tau_i}(t+1) = v_{\tau_i}(t) + \delta_{\tau_i}(t), \qquad \delta_{\tau_i}(t) > 0 \ \forall t, \tau_i \tag{4.3}$$

The property expressed on the right of the formula is crucial for the CFS to ensure non starvation of any runnable task. In fact, under any condition, the virtual runtime of the task currently in execution is always updated with a strictly positive increment at least as often as the scheduler tick frequency (which is 1000 Hz by default for desktop system). Thus, if there are other runnable tasks on the runqueue, the vruntime of the task in execution will eventually increase to not being the minimum anymore and the task will be preempted in favor of another one. This applies at any time and thus each runnable task will be picked up and cannot suffer starvation: this is a desirable property which must not be broken by PAFS.

The update of the vruntime is the chosen point of application of the autonomic action of PAFS; this means that in place of the δ value computed by the CFS there is a new increment value, call it p_{τ_i}, which keeps the property of strict positivity and embeds the knowledge on the applications' performance provided by HRM. The autonomic action of PAFS, with respect to the unmodified CFS, depends on the relationship between p_{τ_i} and δ_{τ_i}; in particular:

- If $p_{\tau_i} < \delta_{\tau_i}$, the task τ_i will be advantaged, as its updated vruntime will be lower than what it would have been with the CFS; thus, it will be kept in execution for a longer time, receiving a larger share of processor time.

- Otherwise, if $p_{\tau_i} < \delta_{\tau_i}$, the task i will be penalized, as it will tend to be executed for a shorter time with respect to the other runnable tasks on the same runqueue.

Clearly, if $p_{\tau_i} = \delta_{\tau_i}$, there is no modification over the default behavior of the CFS.

The new increment must be function of the relationship between the current and the desired heart rates of the group the task τ_i is member of; moreover, p_{τ_i} must still take into account how long the task τ_i has been running for since the last update. Since the focus of this work is not on decision techniques, it has been chosen to look for the simplest possible policy able to satisfy these requirements. Proving the effectiveness of this simple decision mechanism demonstrates the validity of the approach and opens the way to future works focused on providing more sophisticated decision policies for PAFS. The decision mechanism that has been defined in this thesis for determining p_{τ_i} is the heuristic which scales the δ_{τ_i} increment computed by the CFS according to how the current heart rate compares with the desired range. To do so, a simple *performance indicator*, referred to as $\pi_{g(\tau_i)}(t)$ has bee defined to represent the performance, at time t of the group $g(\tau_i)$ the task τ_i is member of as a producer. The formula used to compute this performance indicator is given in Equation (4.4). In the formula, $hr_{g(\tau_i)}$ represents the current heart rate of the group, while $mhr_{g(\tau_i)}$ and $Mhr_{g(\tau_i)}$ refer to the minimum and maximum heart rates.

$$\pi_{g(\tau_i)}(t) = \frac{hr_{g(\tau_i)}(t)}{\bar{hr}_{g(\tau_i)}(t)}, \qquad \bar{hr}_{g(\tau_i)(t)} = \frac{mhr_{g(\tau_i)(t)} + Mhr_{g(\tau_i)}(t)}{2} \tag{4.4}$$

Thus, the performance indicator at a certain time is defined as the ratio between the current heart rate of the group and the average between the minimum and maximum heart rates. This performance indicator is used to scale the CFS-computed increment as in Equation (4.5).

$$p_{\tau_i}(t) = \begin{cases} \frac{1}{S_m}\pi_{g(\tau_i)}(t)\delta_{\tau_i}(t), & hr_{g(\tau_i)}(t) < mhr_{g(\tau_i)}(t) \\ \delta_{\tau_i}(t), & mhr_{g(\tau_i)}(t) \leq hr_{g(\tau_i)}(t) \leq Mhr_{g(\tau_i)}(t) \\ S_M\pi_{g(\tau_i)}(t)\delta_{\tau_i}(t), & hr_{g(\tau_i)}(t) > Mhr_{g(\tau_i)}(t) \end{cases}, \quad S_m, S_M \geq 1 \tag{4.5}$$

As the formula shows, the performance indicator of a group is used to scale the increment of its producers when the current heart rate is outside the desired range, while the δ_{τ_i} is used as-is when the current heart rate is within the minimum and the maximum bounds. In this way, the exact behavior of the CFS is maintained for the tasks of those monitored groups which are yielding the desired Quality of Service, while the increment is modified for the tasks

in those groups which are running either too fast or too slow. Since the performance indicator π_{τ_i} - as shown in Equation (4.4) - is smaller than one when the current heart rate is below the minimum, and it is greater than one when the current heart rate is over the maximum, the increment will be correctly scaled down (advantaging the tasks) or up (disadvantaging them) as it should be. The additional terms S_m and S_M are two parameters used to increase the "strength" of the autonomic action, if needed. When both S_m and S_M are set to 1, the performance indicator is used "as it is" to scale the increment; increasing the values of these two parameters will increase the action on the vruntime update (i.e., setting $S_m > 1$ will yield lower increment values when $hr_{g(\tau_i)} < mhr_{g(\tau_i)}$ and setting $S_M > 1$ will result in higher increment values when $hr_{g(\tau_i)} > Mhr_{g(\tau_i)}$).

4.3 Summary

Within this Chapter, a methodology for the creation of autonomic computing systems, as intended within the CHANGE group, is formalized and illustrated. Particular attention is dedicated, in the second part of the Chapter, to the main contributions carried by this thesis, which regard the proposal of a software monitor and its use to measure applications' performance for realizing performance-aware adaptive scheduling. The design proposed for the proposed monitor (i.e., HRM) is focused on being lightweight while offering the performance statistics to any adaptation policy that may need them. The performance aware process scheduler that has been created for this thesis (i.e., PAFS) is the first of these adaptation policies to be realized and, with its simplicity, serves as a test bench for the whole approach. In the next Chapter, the implementation over the Linux kernel that realizes the design proposed above is illustrated for both the performance monitor and the adaptive process scheduler.

Chapter 5

Proposed Implementation

A first prototype of AcOS, comprising the Heart Rate Monitor, used as an applications' performance monitor and the Performance-Aware Fair Scheduler has actually been implemented over the Linux kernel, following the design principles expressed in Chapter 4. This implementation, which is illustrated in details in this Chapter, is based on Linux-2.6.35.14 [73], which is a *longterm-support* release; the proposed modifications, however, are not specific to this version and can be ported with very little effort to newer kernel releases (as long as no major changes are introduced in the modified portions of kernel code). The rest of this Chapter presents the realized implementation by first showing some general ideas (in Section 5.1) and then focusing on HRM (Section 5.2) and PAFS (Section 5.3), where the latter makes use of the former to gather the information it needs to be aware of the performance of the tasks it must schedule.

5.1 From Linux towards AcOS

The methodology illustrated in Chapter 4 has been employed to implement the proposed monitor (i.e., HRM), which is used to measure applications' performance and the adaptation policy, namely PAFS, which uses the monitor to augment the default process scheduler used in Linux (i.e., the CFS) with performance-awareness. This work is the first step towards the creation of an autonomic layer over Linux and the creation of a Linux-based autonomic operating system.

Linux is a monolithic kernel and it implements the process scheduler fully in kernelspace;

for this reason, the most suitable way of creating the PAFS adaptation policy is to modify the process scheduler within the kernel. As the adaptation policy is to be implemented within the kernel, it is a good idea to build also HRM in kernelspace, to grant easier access for PAFS to the monitoring information. HRM, however, needs to fetch the heartbeats emitted by the instrumented applications, which live in userspace: for this reason an interface from userspace to kernelspace is needed to allow the tasks to emit heartbeats. The creation of a neat and efficient such interface is a key contribution of the implementation part of the work presented in this thesis. Moreover, the proposed implementation takes into account issues such as false sharing in multicore systems and lock contention, granting outstanding performance (i.e., low overhead) to the monitoring infrastructure and simplicity of implementation to the adaptation policy (thanks to the very simple in-kernel availability of the monitoring information). The whole implementation of the autonomic components has been wrapped by using preprocessor macros for conditional compiling and an "*Autonomic Operating System*" menu has been added to the Linux configuration system, with entries to choose whether to enable HRM and/or PAFS or to compile them out, leading to a vanilla Linux. Also all the parameters that are tunable at compile time can be set through a configuration entry (i.e., `CONFIG_*`) and the dependencies between the autonomic components (e.g., PAFS requires HRM) are tracked by the configuration tool. This implementation practices allow great flexibility and modularity and ease the future addition of more autonomic components.

The result of the implementation of HRM and PAFS are two kernel patches (one for each component) and a userspace library, namely *libhrm*, which offers the API to be used for instrumenting the applications to be monitored; the patches and the source code for *libhrm* will be soon made freely available on the website of the CHANGE group [10]. The remaining of this Chapter covers the technical details and documents the implementation.

5.2 Heart Rate Monitor

The implementation of the Heart Rate Monitor follows the design described in Section 4.2.3 focusing on being lightweight (i.e., limiting as much as possible the overhead over the monitored tasks), precise and easy to use.

5.2.1 Overall structure

HRM consists of a patch to the Linux kernel, which enhances it with the new monitoring infrastructure, and a userspace library (named *libhrm*), which exposes an API to be used for instrumenting applications and accessing to the monitoring information from userspace. The relationships between the applications, HRM and the adaptation policies (either in kernel- or userspace) resemble a producer-consumer model where the applications, by emitting heartbeats, produce monitoring information through HRM and the adaptation policies are the consumers using this information. Hence, this terminology is used in the implementation of HRM. Note that HRM supports consumers (i.e., adaptation policies) both in kernelspace (by directly exposing the relevant data structures) and in userspace (through *libhrm*). The support for userspace consumers, however is not directly used in this thesis as the only implemented adaptation policy is PAFS, which works in kernelspace. This capability will be used by future works extending AcOS with different adaptation policies.

A quick overview of the overall structure of the implementation of HRM over Linux can be summarized in the following points:

- The interface between kernel- and userspace has been realized by employing the `procfs` pseudo-filesystem [22] as a firsthand communication channel used to allow mapping shared memory pages both in kernel- and userspace. Producers and consumers can then make use of the *libhrm* API (which wraps in easy-to-use function calls the communication protocol over the `procfs` and the shared memory) to access the respective HRM's functionalities.

- The data structure representing a task within the Linux kernel (which is coded in the source file `include/linux/sched.h` and called `struct task_struct`) has been extended by adding a field with a new data structure containing the monitoring information.

- Each group contains a list of its producers (i.e., its member tasks) and a list of its consumers (i.e., the userspace adaptation policies requesting the monitoring information regarding the group); kernelspace adaptation policies are not tracked by HRM, as they can directly read the fields in the kernel data structures and there is no additional work to be done for letting them access the monitoring information.

These points are expanded and more extensively covered in the next Sections, providing a thorough overview of how HRM has been implemented over the Linux kernel.

5.2.2 A Smart Interface for Producers and Consumers

For HRM to be a useful monitoring utility, it must be easy to use by applications developers and final users, which must be given simple APIs and interfaced for doing so; moreover, the implementation of the communication channels must be fast for high-frequency channels (e.g., heartbeats emission). One of the challenges during the development of HRM was being able to find an efficient way to let the applications issue heartbeats. Since HRM resides in kernelspace, while the instrumented applications are executed in userspace, the emission of a heartbeat must find a way to go across the two different addressing spaces. Moreover, since some applications may be characterized by a hotspot executed at very high frequency (with heart rates of millions of heartbeats per second), a strong requirement is to have a very quick heartbeats issuing mechanism. For this reason, the option of using a system call has been immediately discarded, as it implies a context switch and does not comply with the need of being very quick [52]. As already stated while describing the design of HRM (in Section 4.2.3), it has been chosen to use a shared memory area to allow bidirectional communication between kernel- and userspace. Producers (i.e., monitored tasks) are offered an API to issue heartbeats, while consumers (i.e., userspace adaptation policies, which may also reside within a monitored application itself, in the case of a self-adjusting application) can use library functions to access the monitoring information (i.e., current and goal heart rate).

Shared Memory Creation through `mmap` and `procfs`

The allocation and mapping of the shared memory area must be done on the kernel side, but it must be triggered by a request coming from userspace. For this reason, there is the need of a communication channel open beforehand to allow these requests. A possible way of doing so would be using syscalls, but it has been chosen to avoid introducing new system calls (which, in Linux, need a unique identifier that could clash with new system calls that may be introduced by upstream in future[52]) to maintain the highest possible compatibility with mainline Linux for easing the port of the autonomic layer to newer kernel versions. Instead of system calls, the creation of the shared memory area is based on a mechanism using the `procfs`

pseudo file system [22] and the existing `mmap` system call[22]. Some new files have been added within the `procfs` by modifying the kernel source file `fs/proc/base.c`; these files reside in the folders associated with each existing task (i.e., `/proc/$PID/task/$TID`) and are used by the *libhrm* API implementation to manage the creation of the shared memory channel. Each file is associated a callback function for the different possible file operations from userspace (i.e., read, write, or mmap); when a *libhrm* function performs one of these operations on a `hrm_*` file, it is in fact calling the correspondent kernelspace handler, which manages the request. The `procfs` files are listed in Table 5.1, with a description of what each file operation is used for. These files are used by the *libhrm* functions to initialize the shared memory for holding the

Table 5.1: HRM-related pseudo-files in the `procfs` and their use

FILE NAME[1]	**READ**[2]	**WRITE**[2]	**MMAP**[2]
`hrm_{producer | consumer}_group`	-	Attach to/detach from a group.	-
`hrm_{producer | consumer}_counter`	Read mapping address for counters shared memory.	-	Request access to the counters[3].
`hrm_{producer | consumer}_stats_target`	Read mapping address for statistics and goals shared memory.	-	Request access to the statistics and goals of the group.

[1] The files are contained under the prefix `/proc/$PID/task/$TID`, for each existing task.

[2] Only the task owning the prefix directory (i.e., the task with thread id $TID) is authorized to such operations.

[3] Producers are returned the exact address of their counter and have R/W access; consumers are allowed readonly access.

heartbeats counters, the statistics (i.e., the monitoring information: global and window heart rates), and the group goal. When an userspace task (either an instrumented task in a producer or a task belonging to a userspace adaptation policy) calls the appropriate *libhrm* function to request being attached to a certain group, it passes as parameters the requested group's id (i.e., `$gid`) and a flag indicating whether it is requesting to be attached as a producer or as a consumer (i.e., `$pc`). Then, the communications are managed according to a protocol implemented through *libhrm* and the kernel (in `proc/fs/base.c`), which is represented in Figure 5.1. As the diagram shows, the attach call from the task to *libhrm* is followed by a write of the `$gid` parameter to a file on the `procfs` to pass the request to the kernel. This file is located in the `/proc/$PID/task/$TID` directory corresponding to the process and thread ids

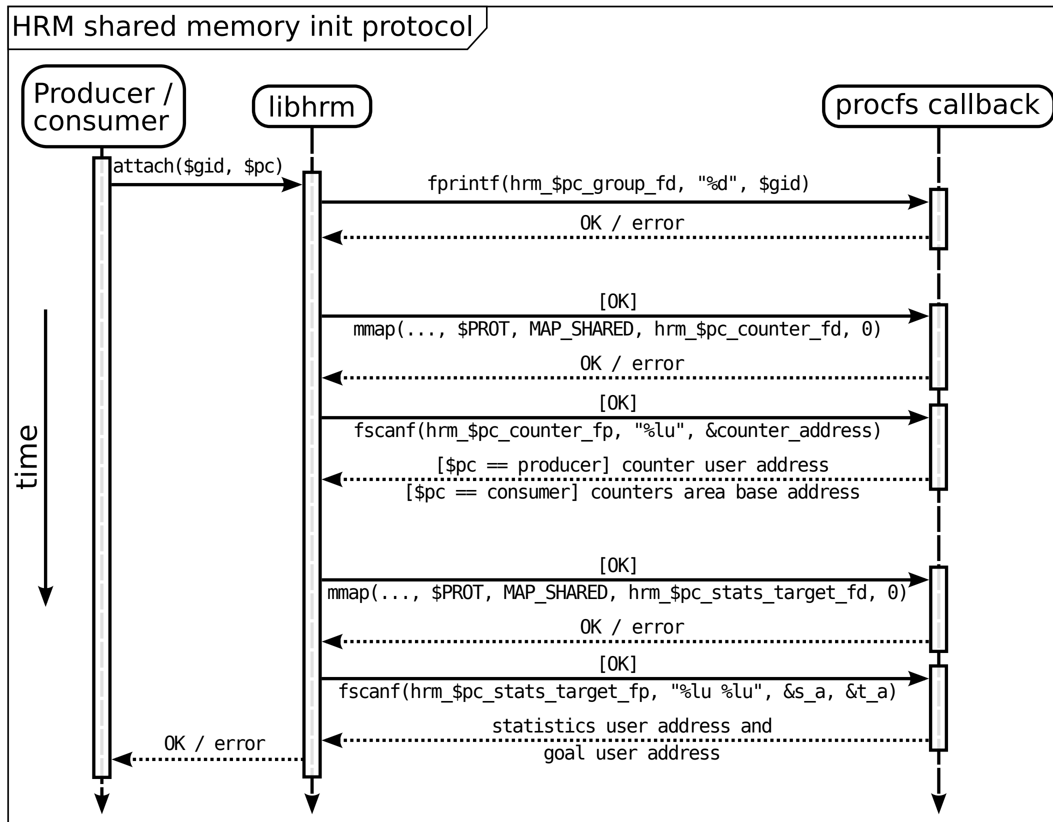


Figure 5.1: Communication protocol used for initializing and mapping the memory areas shared between kernelspace and userspace

of the calling task and it is either `hrm_producer_group` or `hrm_consumer_group`, according to the value of the `$pc` parameter. The callback function implemented on the kernel side passes the request to HRM, which performs some operations to attach the requesting task to the group; the most important steps are the following:

- It makes an authorization check, allowing only the task owning the file in the procfs to make the request (this forbids any task to make a request for another one).
- It gets the `$gid` value and it calls an internal HRM function which attaches the task to the group; this operation involves the following two main steps:
  - allocating the data structures tracking the producer or the consumer;
  - looking for the requested group and, if it does not exist, creating it and allocating the memory to be shared;

- If anything goes in error, it cleans up what was done and returns an error; otherwise, the request was successful and all is set up to permit the mapping of the shared memory.

If the attach request to the kernel was successful, *libhrm* issues the `mmap()` system call on the `hrm_{producer | consumer}_counter` file to request the memory area containing the counters to be mapped onto the caller's address space. If the caller is a producer, the mapping is requested (by means of the `$PROT` parameter) with read/write permissions, as the task needs to emit heartbeats. Consumers need to read the counters memory area because it contains the tids of all the producers members of the group; since they must not write, but only see what tasks are registered, in this case the mapping is read only. The correspondent `procfs` callback function, after having checked that the caller task is already attached to a group and having made some authorization checks, remaps the counters memory area into the caller's address space by means of the `remap_pfn_range()` kernel function. If the operation was successful, the caller has now the counters memory, which was allocated by the kernel, mapped onto its own address space, but the `mmap()` function only returns a page-aligned mapping address, while the the exact address (i.e., page-aligned base + offset) of the counter is needed. The exact address is exposed by the kernel on the `hrm_{producer | consumer}_counter` files; hence, *libhrm* reads the proper file and, depending on whether it was called by a producer or by a consumer, it gets a different address:

- a producer is given the exact address of its counter entry;
- a consumer (which has a read only map) is returned the base address of the counters memory area.

Thus, a producer will be able to emit heartbeats to its counter, while a consumer will have read only access to the whole area. The protocol for mapping the memory containing the group's statistics and goal works exactly in the same way just described for the counters mapping, with the difference that both producers and consumers are given two exact addresses (i.e., base + offset) for the statistics and goals fields.

All these operations are wrapped by the *libhrm* API (which is more extensively covered in Section 5.2.6) and a task just needs to make a single call to the attach function to obtain this fast and convenient shared memory-based communication channel across kernelspace and userspace.

Shared Memory Structure

The memory areas used to allow emitting of heartbeats on one side and exposing statistics on the other are allocated within the kernel in the form of memory pages. The request of free memory pages in the kernel is done by calling the function `__get_free_pages()`, which takes the number of requested pages (which must be a power of two) and returns the address of a block of the requested number of pages (i.e., the pages are physically contiguous in memory). For each HRM group, there are at least two memory pages, used for storing the heartbeats counters, the statistics, and the goals with this organization:

- a minimum of 1 and a maximum of 16 pages are devoted to the counters (this parameter is configurable at compile time with the `CONFIG_HRM_GROUP_ORDER` configuration entry);
- one page is used to store both the statistics and the goals.

The contiguity of the allocated pages is very convenient when more than one page is used to store counters, as the whole memory area is contiguous and there is no need to track the page limits. In Linux, on most architectures, a memory page is 4 KB (i.e., 2^{12} Bytes) long; Table 5.2 shows a representation of the structure of a memory page containing heartbeats counters. The table shows that the counters' memory page is simply an array of slots each containing a data structure representing a producer task registered to the group. For each task, the following fields are tracked:

- the tid of the task owning the counter is stored in `const pid_t tid`;
- the `const int used` tracks whether the slot is or not in use by a task;
- the `int active` signals whether the task is in an *active state* (i.e., if it is executing the hotspot);
- finally, the heartbeats count is maintained in the `uint64_t counter` field.

The next field in the Table, namely, `uint8_t padding[40]`, is interposed between two entries to avoid false sharing issues on multicore processors. The padding is 40 Bytes long to reach, for each block, the size of 64 Bytes, which is the cache-line size on modern x86* processors, which HRM is currently targeting. To support different architectures, the length of the padding

Table 5.2: Structure of a shared memory page used by HRM for counters

OFFSET[1]	**CONTENT**	**DESCRIPTION**
0x000	`const pid_t tid`	1st task identifier.
0x004	`const int used`	Housekeeping fields.
0x008	`int active`	
0x010	`uint64_t counter`	Heartbeats counter.
0x018	`uint8_t padding[40]`	Padding for cache alignment.
0x040	`const pid_t tid`	2nd task identifier.
0x044	`const int used`	Housekeeping fields.
0x048	`int active`	
0x050	`uint64_t counter`	Heartbeats counter.
0x058	`uint8_t padding[40]`	Padding for cache alignment.
⋮	⋮	⋮
0xFC0	`const pid_t tid`	64th task identifier.
0xFC4	`const int used`	Housekeeping fields.
0xFC8	`int active`	
0xFD0	`uint64_t counter`	Heartbeats counter.
0xFD8	`uint8_t padding[40]`	Padding for cache alignment.

[1] The offset is expressed in Bytes.

must simply be adjusted accordingly. The cache-alignment of the counters was initially not considered and was added after some initial experimental evaluations of the monitor overhead, which gave unexpected results due to false sharing problems (see Chapter 6 for more details). With this organization of the counters, each page can hold up to 64 counters and HRM supports groups with 64 counters per used memory page, reaching support for 1024 tasks per group with 16 dedicated memory pages (which is the current maximum allowed, but it is just an arbitrary choice, larger values could be used with no issues).

The information related to the statistics and goals of a group is stored in a separate memory page; its structure is shown in Table 5.3. As shown in the Table, the top part of the page contains two fields which are used to store the group's statistics (i.e., the global and window heart rates); these statistics are periodically computed in the context of a high resolution timer, as illustrated in Section 5.2.5. After the first two fields, there is a padding

Table 5.3: Structure of a shared memory page used by HRM for statistics and goals

OFFSET[1]	CONTENT	DESCRIPTION
0x000	`uint32_t global_heart_rate`	Group's statistics.
0x004	`uint32_t window_heart_rate`	
0x008	`uint8_t padding[56]`	Padding for cache alignment.
0x040	`uint32_t min_heart_rate`	Group's goal heart rates.
0x044	`uint32_t max_heart_rate`	
0x048	`size_t window_size`	Time window size.
0x050	`int64_t timer_period`	Statistics computation period.

[1] The offset is expressed in Bytes.

entry which serves for cache alignment, just as in the counters' pages. The second block contains information related to the group's goal; in particular:

- The `uint32_t min_heart_rate` and `uint32_t min_heart_rate` fields hold the group's maximum and minimum goal heart rates. These values may be changed by the adaptation policies (which may be done directly for kernelspace ones and through *libhrm* for userspace ones) or by the users through a special pseudo-file in the `procfs` (see Section 5.2.3 for more details).

- The time window considered when computing the window heart rate is stored in the `size_t window_size` field. The `CONFIG_HRM_WINDOW_SIZE` kernel configuration entry allows to specify a default window size, which may be tuned at compile time from 128 to 1024 timer periods. Moreover, the window size may be specified at runtime by the producers through *libhrm*.

- Finally, the value of the timer period is held in the field `int64_t timer_period`; this value is configurable at compile time via the `CONFIG_HRM_TIMER_PERIOD` configuration entry.

As already highlighted, this structure of the shared memory pages allows to support applications with groups up to 1024 tasks (but this is just an arbitrary limit) which are assigned a private cache-aligned counter each for lockless heartbeats emission. This features, along with

the asynchronous statistics computation (better illustrated in Section 5.2.5), are some of the key contributions of this thesis in terms of engineering of an efficient general-purpose software monitoring system.

5.2.3 Statistics Visualization and User Control

Other than being useful for the adaptation policies, the statistics computed by HRM (i.e., the global and window heart rates of the monitored groups) are also directly exposed to the users, who can display them. Moreover, the users must be allowed to manually change the goals of groups associated to applications they own; this possibility is of key importance to give the users the power to modify those goals which could be (possibly on purpose) set too high or too low by the applications developers. The possibility of displaying the monitoring information and of changing the maximum and window heart rates is realized through dedicated pseudo-files in the `procfs`, as shown in Table 5.4. The `/proc/hrm` pseudo file is used only for exposing

Table 5.4: HRM-pseudo-files for displaying monitoring information and managing the groups' goal

FILE NAME	READ	WRITE
`/proc/hrm`	Display monitoring information for all the monitored groups.	-
`/proc/$PID/task/$TID/hrm_goal`	Display the group's goal[1].	Modify the group's goals[1].

[1] Only the user owning the corresponding task is authorized to these operations

human-readable information about the monitoring activity HRM is performing. an example of the output returned when reading this file from console is the following (reported on two columns):

```
gid: 37
tids: 2519 2520 2521
global heart rate: 1433
window heart rate: 1439
minimum heart rate: 1300
maximum heart rate: 1500
window size: 125
timer period: 100000
```

```
gid: 42
tids: 2516 2517
global heart rate: 748
window heart rate: 799
minimum heart rate: 780
maximum heart rate: 1000
window size: 50
timer period: 100000
```

The pseudo-file reports the current monitoring information reported by HRM. In the example, there are two monitored groups (with gids 37 and 42), which are currently tracking the prgress

of three and two producer tasks respectively. For each group, the current and the goal heart rates are shown, together with the window size and the timer period. These data can be useful for applications developers, as a debugging tool during applications instrumentation or to users and system administrators to monitor the current state of the applications in execution monitored with HRM.

The `/proc/$PID/task/$TID/hrm_goal` can be used only by the user who owns the corresponding task (i.e., the one with tid equal to `$TID`). When reading this file, the current goal of the group the task is member of (as a producer) is displayed (or an error is returned if the task is not currently monitored). By writing to this file (e.g., from the shell, with `echo $CMD > /proc/$PID/task/$TID/hrm_goal`), the users can change the current goal by using the following commands:

- with the "`m$VALUE`", it is possible to set the new minimum heart rate to `$value`;
- to set the maximum heart rate to a `$VALUE`, the command is "`M$VALUE`".

The `procfs` callback function makes some simple checks (e.g., it verifies that the maximum heart rate is greater than the minimum heart rate) and modifies the goal for the group the task associated with the used file is member of as a producer.

5.2.4 Structure of a Group within the Kernel

On the kernel side, a group of tasks monitored with HRM is represented with a dedicated data structure added in the new kernel header file `include/linux/hrm.h`. This source file contains all the definitions of the new data structures introduced with HRM and the prototypes of the kernelspace API used within the `procfs` callback functions to interact with HRM. The data structure used to represent a group is named `struct hrm_group` and it is reported in Listing 5.1, with some comments describing the different fields.

```
1  struct hrm_group {
2      int gid;                           /* group's identifier                              */
3      struct hrm_memory counters;        /* manage shared memory for counters               */
4      struct hrm_memory stats_target;    /* manage shared memory for stats and goals        */
5
6      struct {
7              int window_begin;          /* index of first heartbeats snapshot              */
8              int window_end;            /* index of last heartbeats snapshot               */
9
10             struct {
11                 u64 counter;           /* heartbeats counter snapshot                     */
12                 struct timespec
13                         elapsed_time;  /* timestamp of this heartbeats count              */
14             } window[HRM_WINDOW_SIZE]; /* array used as circular buffer for tracking
15                                           the last HRM_WINDOW_SIZE heartbeats counts      */
16
17             u64 history;               /* field for keeping dead producers' counters      */
18     } history;                         /* keep track of heartbeats history                */
19
20     DECLARE_BITMAP(counters_allocation,
21                     HRM_GROUP_SIZE);   /* bitmap to track the used counter slots          */
22
23     struct hrtimer timer;              /* hrtimer for statistics accounting               */
24     struct timespec timer_period;      /* hrtimer activation period                       */
25     struct timespec elapsed_time;      /* time since group was created                    */
26     struct timespec timestamp;         /* group creation timestamp                        */
27
28     struct list_head producers;        /* list of registered producers                    */
29     struct list_head consumers;        /* list of registered consumers                    */
30     rwlock_t members_lock;             /* group members lock                              */
31
32     struct list_head link;             /* link to next group in the groups list           */
33 };
```

Listing 5.1: Data structure representing a HRM group within the Linux kernel (defined in `include/linux/hrm.h`)

The new data structure contains all the group-related information. The first field is the group's identifier, which is unique in HRM (i.e., if a task requests to be attached to the `gid` of an existing group, it is added to that group and there is no means of creating a new one with the same identifier). The next two fields are used to keep track of the memory pages allocated for the counters and the statistics and goals; the type of these fields is another data structure added by HRM: the `struct hrm_memory`, reported in Listing 5.2.

```
1 struct hrm_memory {
2     unsigned long kernel_address;   /* kernelspace address of the tracked memory area      */
3     size_t size;                    /* size of the memory area at kernel_address           */
4     struct list_head maps;          /* list of attached userspace mappings                 */
5 };
```

Listing 5.2: Kernelspace data structure (defined in `include/linux/hrm.h`) for tracking allocated shared memory pages

The `kernel_address` field stores the kernelspace base address of the memory area. Remember that, if more than one page is used for the counters, the whole memory area is nonetheless contiguous, so it is enough to keep the base address and the size of the area (stored in the `size` field). The last field of the `struct hrm_memory` is used to keep track of which userspace processes have the correspondent memory area mapped to their address space. This list (see Love [52] for how linked lists) links to entities of type `struct hrm_memory_map` (reported in Listing 5.3), which are used to store the mapping-related information.

```
1 struct hrm_memory_map {
2     pid_t pid;                      /* pid of the process owning the mapping               */
3     unsigned long user_address;     /* userspace address of the mapped memory              */
4     int references;                 /* tasks within the thread group using the mapping     */
5     struct list_head link;          /* link to the next memory map in the list             */
6 };
```

Listing 5.3: Kernelspace data structure (defined in `kernel/hrm.c`) for managing memory mappings

For each existing mapping, HRM stores the `pid` (i.e., the process identifier) corresponding to the address space where the memory is mapped. Recall that, in Linux (as in most *NIX systems), the address space is shared among a process and the threads in the same groups (i.e., with that process as their parent) and the thread group is identified by the `pid`. The next field stores the userspace base address of the mapping and the `references` field counts

how many tasks in the thread group are using the mapped memory. Finally, the last field of the struct hrm_memory_map links to the next structure in the list.

Going back to the struct hrm_group (refer to Listing 5.1), the next field after the structures for managing the shared memory pages is a nested data structure accessible from within the group as history. This structure is used to keep the group's history, i.e., to manage the time window for computing the window heart rate (which is done by keeping the window_begin and window_end indexes to use the window[HRM_WINDOW_SIZE] array as a circular buffer) and to keep track of the heartbeats counters of terminated tasks, which must still be considered for correctly computing the global heart rate. More details regarding the computation of the statistics can be found in Section 5.2.5.

After the structures for managing the counters' history, the next field is a kernel macro for declaring a "counters_allocation" bitmap used to keep track, on the kernel side, of the allocation of the counters in the shared pages to the producers that register to the group. This field is simply a bitmap counting as many bits as the maximum number of manageable counters. When the bit in position n in the bitmap is set, this means that the counter slot in position n in the counters' shared memory is in use (and the corresponding active field in the page - as represented in Table 5.2 - is set).

The next block of fields of the struct hrm_group is devoted to the management of the high resolution timer used for computing the statistics and to keeping track of the elapsed time since the group was created and the last field is a link to the next group in the global groups list, which is defined in kernel/hrm.c as shown in Listing 5.4.

```
1 LIST_HEAD(hrm_groups);                /* declaration of the global groups list           */
2 DEFINE_SPINLOCK(hrm_groups_lock);    /* spinlock for protecting the groups list          */
```

Listing 5.4: Declaration of the global HRM groups list (found in kernel/hrm.c)

A spinlock is declared along with the groups list, and it is used for locking it to avoid concurrency issues when adding and removing groups. Moreover, both the list and the spinlock are declared as extern in the header file include/linux/hrm.h and thus both are globally accessible from anywhere in the kernel.

The block of fields over the list hook in struct hrm_group are used for keeping track of the producers and consumers currently attached to the group. In particular, there are two

lists, respectively linking to producers and consumers, and a lock to protect the lists. Only one lock is used to protect both lists to reduce the locking complexity, since some experiments showed that using two locks did not ensure big performance improvements but primarily made the locking code more complex to use and maintain. The lists of producers and consumers link entities of respective types `struct hrm_producers` and `struct hrm_consumers`, which are reported in Listing 5.5.

```
1  struct hrm_producer {
2      int counter_index;                 /* index of the counter assigned in the shared page */
3
4      struct hrm_counter *counter;       /* pointer to the counter address                   */
5      struct hrm_stats *stats;           /* pointer to the group's statistics                */
6      struct hrm_target *target;         /* pointer to the group's goal                      */
7
8      struct hrm_group *group;           /* back reference to the group                      */
9
10     unsigned long counter_user_address; /* userspace mapped address of the counter         */
11     unsigned long stats_user_address;   /* userspace mapped address of the statistics      */
12     unsigned long target_user_address;  /* userspace mapped address of the goals           */
13
14     struct list_head link;             /* link to the next producer in the list            */
15 };
16 struct hrm_consumer {
17     struct hrm_group *group;           /* back reference to the group                      */
18
19     unsigned long counter_user_address; /* userspace mapped base address of the counters   */
20     unsigned long stats_user_address;   /* userspace mapped address of the statistics      */
21     unsigned long target_user_address;  /* userspace mapped address of the goals           */
22
23     struct list_head task_link;        /* used for tracking the groups a consumer task is
24                                           attached to                                      */
25     struct list_head group_link;       /* link to the group's producers list               */
26 };
```

Listing 5.5: Kernelspace data structures for HRM producers and consumers (declared in `include/linux/hrm.h`)

These data structures basically contain references to the HRM data needed by the producer/consumer. In particular, the `struct hrm_producer` contains a link to the next entity in the producers list (the head of this list is the `struct list_head producers` field in the `struct hrm_group`). A similar field is present i`hrm_detach`n the `struct hrm_consumer`, where the `struct list_head group_link` field has this same role, linking to a list with head in

the `struct list_head consumers` field of the `struct hrm_group`. Moreover, in the `struct hrm_consumer` there is another list (i.e., `struct list_head task_link`), which is used to keep track of what groups a consumer is attached to. This is done by placing the head of this list in the `struct task_struct`, which is defined in `include/linux/sched.h` and is the data structure used by Linux to keep track of all the runtime information related to a task. In particular, the additions made to this structure to support HRM are shown in Listing 5.6.

```
1  struct task_struct {
2      ...
3      struct hrm_producer hrm_producer;   /* used if the task is an active HRM producer      */
4      struct list_head hrm_consumers;     /* used for tracking the observed groups in case
5                                             the task is active as a HRM consumer            */
6  }
```

Listing 5.6: Additions to the `task task_struct` (in `include/linux/sched.h`) to support HRM producers and consumers

The two additions are placed at the end of the task structure and are used in this way:

- The first field is used to keep the information needed to the task when it is active as a producer. Since a task may be attached to only one group at the same time, a list is not required here and a single data structure of type `struct hrm_producer` (see Listing 5.5) holds all the information regarding a producer task.
- Since a consumer may be attached to more than one group at the same time, to track all the groups a consumer is attached to a list is needed. The second field in Listing 5.6 is the head of such list, which is linked to structures of type `struct hrm_consumer` through the field `task_link` (refer to Listing 5.5).

Thus, each entity of type `struct hrm_consumer` acts as an interface between a task and the groups it is attached to as a consumer, allowing a group to know what consumers are currently attached to it and, at the same time, a task to know what groups it is attached to. To do so, this structure is contained, as already clarified, in two lists.

The discussion proposed above, covers the most relevant data structures at the base of the organization of a group. This structure is not trivial and there are some interesting interactions between the various entities. To clarify how these occur, Figure 5.2 shows a diagram of the organization of a group. In the Figure, only some fields of the various data structures are

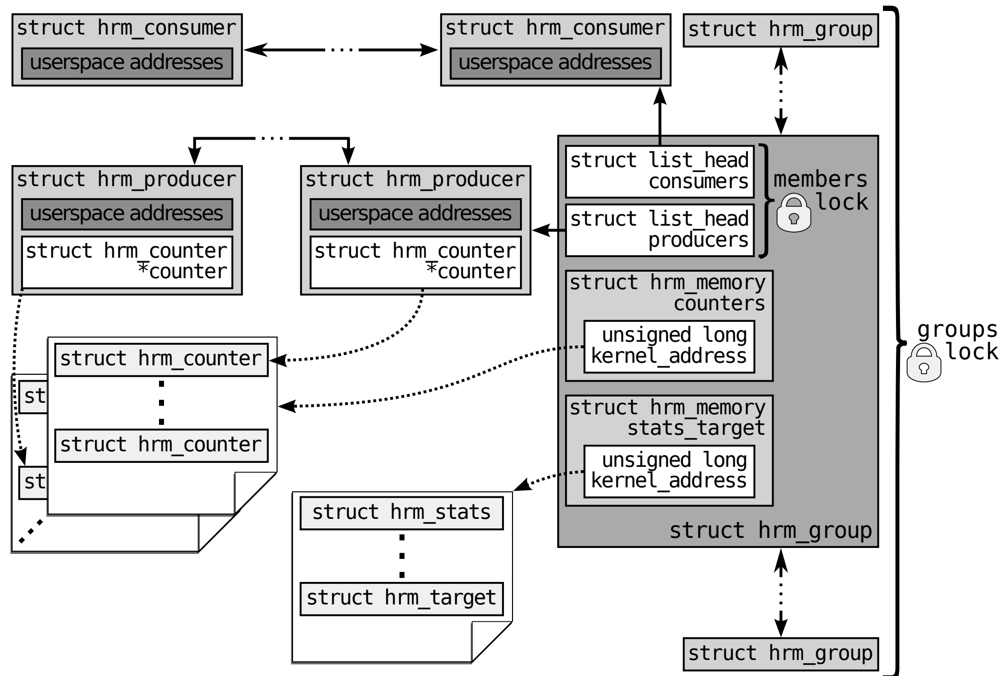


Figure 5.2: Structure of a HRM group

reported, and the role of the lists with head in the group's data structure are highlighted. Also the presence of the global groups list is reported in the Figure, which highlights the role of the locks at protection of the global groups list (see Listing 5.4) and of the group's members lists (see Listing 5.1); the list of consumers with head in the `struct task_struct` is not shown. The areas in the bottom left corner represent the shared memory pages used for the counters and the statistics and goals.

As already highlighted in Section 5.2.2, the structure at the base of the HRM's groups allows completely lockless heartbeats emission and makes this operation very efficient thanks to the shared-memory architecture. The next Section completes the overview of the kernel-side implementation of HRM, presenting how the statistics are computed.

5.2.5 Statistics accounting

According to the proposed design for HRM, the primary monitoring information it provides are the global and window heart rates of the monitored groups. The computation of these statistics is decoupled from the emission of heartbeats by the instrumented tasks and it is managed by

a kernelspace routine which is periodically awaken by a timer; this timer is realized with a high-resolution timer (hrtimer), which ensures good accuracy even with high system load [52]. As already mentioned when describing the structure of a group, the period of computation of the statistics is a parameter of each group and it is stored in the `timer_period` field of its shared page used for storing the statistics. In the current implementation, this value cannot be changed at runtime (i.e., all the groups will keep the default period), but allowing to set a timer period specific for each group, if needed, is a straightforward extension. The fixed value for the timer period can be configured at compile time by modifying the `CONFIG_HRM_TIMER_PERIOD` configuration entry; the supported values are 1, 10, 100, and 1000ms and the default value is 100ms. As for the number of pages used for the counters, also these values are arbitrary and there is no real limitation to adding support for different periods. Setting a period too long,however, could imply a too low sampling frequency, making the monitor imprecise in catching rapid variations of the heart rates. Setting the period to short, on the other hand, reduces the possible duration in time of the time window used for computing the window heart rate (currently, a value up to 1024 timer periods is supported) and might increase the overhead due to more frequent execution of the statistics computation routine. The supported values are believed to cover a wide range of needs.

The timer period rules how often the statistics (i.e., the global and window heart rates) are computed, which is done once per period; the computation of the statistics proceeds as follows:

- The global heart rate is simply computed as the total sum of the emitted heartbeats over the elapsed monitoring time (as expressed in Equation (4.1)).
- The computation of the window heart rate is performed as shown in Equation (4.2) by using the array which elements are defined in the structure nested in the `struct hrm_group` (and reported in Listing 5.1). This array, which length can be set at compile time with the `CONFIG_HRM_WINDOW_SIZE` configuration entry, is used as a circular buffer to store, at each timer awakening, a snapshot of the current group heartbeats count and the current timestamp. The two indexes (namely, `window_begin` and `window_end`) found in the `history` data structure nested into the `struct hrm_group` are used to keep track of the entry in the circular buffer holding the heartbeats count at the beginning of

the time window and of the entry containing the current heartbeats count. The update of these indexes can be done very efficiently (with a quick bitwise `AND`) if the buffer size is a power of two; for this reason all the possible configurable sizes for the array size are powers of two.

The current time, used to update the time accounting fields in the HRM's data structures which are needed to compute the statistics, is got by means of the `getrawmonotonic()` function, which returns the current raw kernel timestamp. Since the current timestamp is retrieved at each timer activation, the heart rates computed by HRM make use of the most accurate timing information available within the kernel and thus offer a pretty precise accounting.

A preliminary evaluation to support this claim has been realized through an ad-hoc microbenchmark which launches a simple multithreaded application which does nothing but emit a desired number of heartbeats in a loop and then terminates. The application has been executed over an idle system, so that its heart rate wold not be altered by contention with other applications and could be determined by placing a `usleep()` instruction of the desired period within the hotspot. Before exiting, the benchmark computes its actual heart rate with double floating point precision by dividing the number of emitted heartbeats by the actual execution time and compares this value with the measure provided by HRM. The test was run several times with different combinations of throughputs, number of emitted heartbeats, and number of threads; the measure HRM always proved as precise as possible, resulting exactly the nearest smaller integer value to the real heart rate: thus, the maximum error of the measure is of $0.\bar{9}\frac{hb}{s}$. This error comes from the choice of not using floating point arithmetic within the kernel (which is a troublesome and discouraged practice [52]), but sticking to integer operations; this means that HRM is not able to measure sub-unit heart rates (i.e., applications emitting less than a heartbeat per second). To partially overcome this limitation, the *libhrm* API (which is illustrated in Section 5.2.6) offers a `heartbeatN()` function, which emits N heartbeats at once.

The impact of the truncation of the heart rate measure done by HRM on its accuracy clearly depends on the value of the heart rate itself. The relative error is maximum when the real heart rate is of $0.\bar{9}\frac{hb}{s}$ and just one heartbeat is emitted per iteration: in this extreme case, HRM would yield a constant measure of $0\frac{hb}{s}$, with a 99% relative error. The situation, however, gets better as the heart rates increase, as the maximum error decreases as $\frac{0.\bar{9}}{real_heart_rate}$ (for

instance, it is of 25% when the measured heart rate is of $4\frac{hb}{s}$, and it drops below 10% with a measured heart rate of $10\frac{hb}{s}$). This analysis shows that HRM is pretty precise when dealing with reasonably high heart rates, but it is not well suited to monitor applications with very low heart rate. This is a caveat to who instruments applications to do so in a proper way, so as to be sure that a suitable number of heartbeats will be emitted under any use case. By doing so, it is possible to make the error on the measure almost as small as desired (just paying attention to avoid overflows in the counters).

5.2.6 API for Producers and Consumers

Kernelspace adaptation policies can make use of HRM simply by interacting with its data structures (in particular, with the global list of groups and with the fields added to the `struct task_struct` - see Section 5.2.4). The applications to be instrumented and userspace adaptation policies, on the other hand, need an interface letting them interact with the kernel. This interface is realized by the userspace portion of HRM, which is named *libhrm* and is intended for offering a simple yet effective API for producers (i.e., applications to be instrumented) and consumers (i.e., userspace adaptation policies). The API exposed by *libhrm* (and defined in the file `hrm.h`) is reported in Table 5.5. As the table shows, the API counts a handful

Table 5.5: *libhrm* userspace API for producers and consumers[1]

Function	Parameters	Description
`heartbeat`[2]		Emit a heartbeat
`heartbeatN`[2]	`n: unsigned int`	Emit n heartbeats
`hrm_attach`	`gid: int`, `consumer: bool_t`	Attach the task to a group
`hrm_detach`		Detach from group
`hrm_set_{active|inactive}`[2]		Set the task active or inactive
`hrm_set_{min|max}_heart_rate`[2]	`{min|max}: uint32_t`	Set the minimum or maximum heart rate
`hrm_set_window_size`[2]	`size: size_t`	Set the sliding window size
`hrm_get_{global|window}_heart_rate`		Get the global or window heart rate
`hrm_get_{min|max}_heart_rate`		Get the min. or max. heart rate
`hrm_get_{window_size|timer_period}`		Get the window size or the timer period

[1]every function receives an additional parameter of type `hrm_t *`,which is the underlying data structure
[2]consumers (i.e., adaptation policies) cannot call this functions

of functions which expose to both producers and consumers the respective functionalities of HRM. The implementation of these functions is based on variations of the protocol described in Section 5.2.2 and used by the `hrm_attach()` function to register a task to a group.

The typical workflow of an instrumented task using HRM as a producer begins by calling the `htm_attach()` function to register to a group; the function takes as parameters the desired group id and a flag indicating whether the task is a producer or a consumer. Then, the task may change the goal of the group it has been attached to by using the `hrm_set_{min|max}_heart_rate()` functions and it can set the window size to be used for computing the window heart rate by means of the `hrm_set_window_size()` routine. If no window is set, HRM simply does not compute the window heart rate, yielding only the global heart rate. When the task begins executing the hotspot, it must call the `hrm_set_active` function to notify HRM that it is ready to emit heartbeats and, from now on, it can use the `heartbeat()` or the `heartbeatN()` functions to emit heartbeats. At the end of the execution of the hotspot, the task should notify HRM by calling the `hrm_set_inactive()` routine and, prior to exiting, it can call `hrm_detach()` to cleanup its internal data. The last two calls in the workflow, however, are not compulsory, as hooks to the cleanup code have been added in the `kernel/exit.c` kernel source file, so that, if not already done, the cleanup phase is automatically executed upon exit.

Also a userspace adaptation policy would begin the interaction with HRM by calling the `hrm_attach()` function, this time specifying the consumer flag. A consumer, however cannot call some of the functions in the API (i.e., those marked with [2] in Table 5.5); basically, it can just get the monitoring information regarding the grouped he registered to, which is all it needs. As already said, there are currently no userspace adaptation policy implemented in AcOS but, as shown, HRM already fully supports this possibility and is ready for future developments in this direction.

5.3 Performance Aware Fair Scheduler

The monitoring infrastructure created with HRM finds a first use in the realization of a performance aware adaptive scheduling ODA control loop, which is bases its observation phase on HRM and its decision and action phases on PAFS. As already pointed out in Section 4.2.6, when explaining the design chosen for PAFS, it has been chosen to keep this first adaptation policy as simple as possible, avoiding complex decision techniques and adopting a simple heuristic. Due to the well defined infrastructure offered by HRM, it has been possible to

follow this conduct also in the implementation phase of PAFS: having direct access to the monitoring information allowed to implement the heuristic with a tiny patch counting few lines of added code to the kernel source file `kernel/sched_fair.c`.

5.3.1 Plugging the Heuristic into the CFS

The Linux kernel function which is in charge of updating the virtual runtime of the tasks is called `update_curr()` and it is implemented in the source file `kernel/sched_fair.c`. This function retrieves the amount of time the current task has been executed for since the last update and it passes it to a helper function, called `__update_cur()`, which weighs this value by means of the `calc_delta_fair()` function, which computes the final vruntime increment value, which is referred to as δ_{τ_i} in Equation (4.3). The point of application of the adaptation policy has been identified right after the weighted vruntime is computed by the CFS. At this point, a call another helper function called `__calc_delta_pafs()` has been introduced; this function actually implements the heuristic for weighing the vruntime increment according to the current performance of tasks registered as producers in a monitored group. To do so, it performs the following steps:

- First, it checks if the task is an active producer; if it is not, it simply returns the unmodified value computed by the CFS.
- If the task is actually an active producer, the function retrieves the goal and statistics of its group through the `hrm_producer` field in its `struct task_struct`.
- Then, the function computes the performance indicator as in Equation (4.4) and implements the heuristic expressed in Equation (4.5). If for the current group, the window heart rate is available (i.e., if the size of the time window is set to a value greater than one), this measure is used for the current heart rate, otherwise the global heart rate is considered.

Note that inserting the computation of the new vruntime update in this place means that the vruntime will be always updated with the most recent performance information available and that it will not be updated more frequently than needed. This strategy allows to put a very tiny overhead (as the required computations are not really onerous) whenever the vruntime

would have been updated in any case and it avoids the need of going through the list of groups to retrieve the monitoring status.

Thanks to the well integrated structure of HRM within Linux and to the simplicity of the heuristic, the implementation of the performance aware adaptation policy has been quite easy, with the main difficulty in finding the best application point within the implementation of the CFS. The main advantage of using this simple heuristic is that it is possible to seamlessly plug it into the CFS in the exact point where the vruntime was already updated. More complex policies (e.g., an adaptive controller based on identification techniques [51]) may be smarter than this heuristic, but would require the creation of a more complex infrastructure to compute the p_{τ_i} at regular intervals, which poses some integration problems with the dynamic ticks used in the CFS.

5.4 Summary

The implementation of the first prototype for AcOS is presented in this Chapter. This first version of AcOS featurs a software monitor (i.e., HRM) and an adaptation policy (i.e., PAFS) using the monitor to measure applications' performance and realizing a performance aware adaptive scheduling control loop. HRM offers a complete infrastructure for computing the heart rate of the instrumented applications, supports both userspace and kernelspace adaptation policy, and allows the users to set the goals and display the statistics. Thanks to the functionality of HRM and to its good integration within Linux, the implementation of PAFS has been eased, consisting in a small addition to the CFS's source code. The next Chapter validates this implementation, by testing both the performance monitor (and comparing it with AH) and the overall performance aware scheduling capabilities of the proposed system.

Chapter 6

Experimental Results

The results of experimental evaluation of the implementation proposed in Chapter 5 are illustrated in this Chapter. First of all, the experimental environment is illustrated in Section 6.1 in terms of both the hardware platforms and the software tools and benchmarks employed. Then, an overview of all the tests that have been done is given in Section 6.2. HRM has been evaluated in terms of efficiency (i.e., monitoring overhead) and functionality in characterizing a real workload; the results of this evaluation are reported in Section 6.3. Then, the whole performance aware adaptive scheduling control loop, comprising HRM and PAFS, has been evaluated on real workloads and under different conditions; the results of these experiments are shown in Section 6.4.

6.1 Experimental Environment

In order to evaluate and characterize the behavior of the proposed system under different points of view and in different contexts, an articulated environment has been chosen to realize the experiments proposed in this Chapter. The experimental environment comprises three different hardware platforms, a microbenchmark and a real application which has been instrumented with *libhrm*.

6.1.1 Hardware Platforms

To evaluate the efficiency of HRM as a software monitor, two different platforms have been chosen equipped with different processors offering on the same ISA (i.e., `x86_64`), but based

on different microarchitectures:

- Platform A is a workstation featuring a Intel®Pentium™D 820 dual-core microprocessor (based on the *Smithfield* microarchitecture) clocked at 2.80GHz with 1MB of Last Level Cache (LLC) per core and, as the main memory, 1GB of DDR2-800.
- Platform B is a workstation equipped with a more recent quad-core Intel®Core™i7-870 microprocessor (based on the *Lynnfield* microarchitecture) clocked at 2.93GHz with 8MB of shared LLC and, as the main memory, 4GB of DDR3-1066.

The main difference between the two platforms (apart from the age), is that the processor of platform A does not have shared Last Level Cache among the cores and, for this reason, a false-sharing issue would result in a greater performance drop than in a microarchitecture based on a shared LLC design (as the microprocessor of platform B). As it is shown when presenting the results, the first platform has been used during the development of HRM to evaluate possible performance issues of this kind. The processor of platform B also supports the Hyperthreading™technology for Simultaneous MultiThreading (SMT) and the TurboBoost™but some preliminary experiments showed that these advanced features introduced noise in the experimental data, and have thus been disabled during the tests.

Due to some logistic issues unrelated with this work, during the tests it has been needed to substitute platform B with another similar workstation, which has been used for the remaining experiments. This workstation, referred to as platform C, is equipped with a quad-core Intel®Core™i5 750 microprocessor (based on the same *Lynnfield* architecture of the CPU found in platform B) clocked at 2.67GHz and, as the main memory, 4GB of DDR3-1333.

The setup of the three platforms is quite representative of current desktop, laptop or low-end server systems that could benefit from the autonomic features proposed by AcOS by simply installing a properly patched Linux kernel. A validation of the proposed system on different architectures not easily supported by Linux (such as some mobile devices) could require modifications to the implementation (for instance an additional userspace library to permit the use of HRM from within the dalvik virtual machine which is used by Android [43] over Linux) and is left for future works.

6.1.2 Software Test Bench

The mass storage of all the three testing platforms has been formatted and a netinst version of Debian [11] *Squeeze* has been installed. The default Linux kernel shipped with Debian has been substituted with Linux version 2.6.35.14 patched with the AcOS according to the implementation described in Chapter 5.

Two different applications have been used to evaluate HRM and PAFS:

- An ad-hoc microbenchmark, called *tachycardia* has been created to evaluate the monitoring overhead imposed by HRM. As the name suggests, this application does nothing but trying to emit heartbeats as fast as possible.
- To evaluate HRM and PAFS on a real workload, the *x264* video encoder [57] has been instrumented with *libhrm*.

Tachycardia is a simple microbenchmark and its only purpose is to evaluate how fast HRM is able to record heartbeats. On the other hand, x264 is a widely used video encoder presenting some characteristics which make it a suitable test bench for the proposed system:

- The kernel of the video encoding process works as a loop within which the video frames are subsequently encoded: this workload is well suited to be instrumented with HRM.
- x264 is used within the popular *parsec* benchmark suite [74] (in fact, the version of the encoder used for the tests is not taken from parsec, but directly downloaded from the reference website [57]).
- The source code of x264 is freely available online [57].
- x264 supports different *presets* (from *placebo* to *ultrafast*), which make the encoding process faster or slower by affecting the quality of the resulting video. This possibility allows to choose a reference video and run different instances of x264 with different presets to have applications with different heart rates running simultaneously.

For the experiments involving x264, the chosen reference video to encode is "*Big Buck Bunny*", which is freely available online [9], *MP4* version at 1920x1080 pixels *HD* resolution; the task of *x264* is to convert this video to the *H.264* compressed format. The encoding process, in

the used version of x264, is realized by using a thread pool (with a desired number of theads), which manages the encoding of the frames. The instrumentation simply creates one group per encoder and all the working threads are added to the group, emitting one heartbeat for each generated frame and hence contributing to the heart rate of the group. In the experiments, except where differently indicated, x264 is run with 4 threads (as many as the cores of platform C, where the tests involving x264 have been performed).

For some tests, there was the need of simulating a condition of high system load; to do so, the *CPU burn* [60] tool, which is designed to heavily load CPUs, has been used. In particular, in the experiments involving CPU burn, the used program is `burnP6`.

The results concerning x264 have been recorded by exploiting the `/proc/hrm` pseudo-file, through which HRM exposes the monitoring information to the users. A shell script was used to log the contents of this file once per second; then, the raw data have been parsed to draw the graphs.

6.1.3 Experimental Parameters

Prior to gathering the data proposed in this Chapter, some preliminary tests have been run to identify reasonable values for the compile-time tunable parameters for PAFS and HRM; as the optimization of the system parameters is not the focus of this work, these experiments are not very interesting and are omitted from the proposed results. Unless diversely indicated, all the proposed experiments have been run with the settings shown in Table 6.1. The actual

PARAMETER	VALUE	DESCRIPTION
`HRM_TIMER_PERIOD`	100ms	Period for statistics computation
`HRM_WINDOW_SIZE`	1024	Maximum length for the time window
S_m	2	PAFS additional scaling factor when $hr < mhr$
S_M	3	PAFS additional scaling factor when $hr > Mhr$

Table 6.1: System parameters used for the experiments run to obtain the presented results

duration of the time window can be set at runtime and it has been tuned for the chosen workload (i.e., the encode of the Big Buck Bunny *HD* video from MP4 to H.264 by using x264). The effects of changes to this parameter on the provided statistics is more interesting

for the scope of this work and are further analyzed in Section 6.3.2.

6.2 Performed experiments

The results presented in this Chapter are related to six different experiments; three concerning only HRM and the other three also involving PAFS. More in details, results for the following experiments are proposed:

- The experiments regarding HRM alone are illustrated in Section 6.3; these include an analysis of the (positive) impact of the cache-alignment optimization (see Section 5.2.2), the evaluation of the monitoring overhead (by means of tachycardia), with a direct comparison with a state of the art software monitor, and the characterization of the runtime behavior of a run of x264 with different settings for the window size.
- The results of the experiments involving PAFS are presented in Section 6.4; the three proposed experiments concern trying to improve the Quality of Service against an external load, diversifying the performance of two homogeneous instances of x264, and inverting the performance of two heterogeneous (i.e., with different presets) instances of the video encoder.

The discussion proposed in remaining of this Chapter shows and analyzes the results of the performed experiments.

6.3 Heart Rate Monitor

The Heart Rate Monitor has been evaluated to determine how efficient it is; i.e., how much performance overhead it adds when used to monitor applications. The design and implementation of HRM is focused on being as lightweight as possible (e.g., by allowing fully lockless heartbeats emission) and it is designed so that its impact on the performance of the system is null if no monitored group is currently being executed. An analysis of the results provided by HRM in term of efficiency is proposed in Section 6.3.1. The second experiment involving HRM is its use to characterize the execution of an instance of x264 encoding the reference video; the results of this test are proposed in Section 6.3.2.

6.3.1 Monitoring Overhead

To evaluate the overhead of the performance monitor over the monitored applications (i.e. the overhead of emitting heartbeats), a simple approach would be to launch the same application instrumented and not instrumented and to compare the subsequent execution times. This approach, however, highly depends on the heart rate of the application. A more general, and synthetic, way of measuring the monitoring overhead is to shrink as much as possible the work carried on within the hotspot, leaving only the computations needed to emit the heartbeats and compute the statistics. Pushing this idea to its limit, the best application to measure only the monitor overhead is one that does nothing but emitting heartbeats at the highest frequency possible. The custom microbenchmark called tachycardia has been realized according to this idea: it does nothing but create a desired number of threads, attach to a group and start emitting heartbeats at the highest frequency possible, equally dividing the workload among the threads. By using tachycardia, the monitor overhead can be measured as the time (or the number of clock cycles) needed for it to be executed. Clearly, the lower this value or, conversely, the higher the throughput, the better (i.e. the lower the monitoring overhead). Both the experiments proposed in this Section have been recorder without additional system load.

The first experiment performed using tachycardia is an evaluation of the benefits ensured by the cache-alignment padding added in the shared memory pages (refer to Section 5.2.2). This experiment has been realized on platform A, which features the dual-core processor with private per core LLC. This platform has been chosen because the characteristics of its microarchitecture emphasize the negative effects of false-sharing issues. The experiment consists in measuring the heartbeats throughput HRM is able to provide with and without the optimization. To do so, tachycardia has been run multiple times with a different of threads to emit one million heartbeats. The execution time has been recorded and the throughput has been analytically computed as $\frac{1000000}{execution\ time}\left[\frac{hb}{s}\right]$. Figure 6.1 presents the results of this test. As shown by the plots, the version of HRM without the cache-alignment optimization provides a lower throughput with respect to the optimized version (which has been used for all the other experiments). Moreover, the optimized version scales as expected with the number of threads, as it reaches the peak throughput with two threads (recall that the processor of

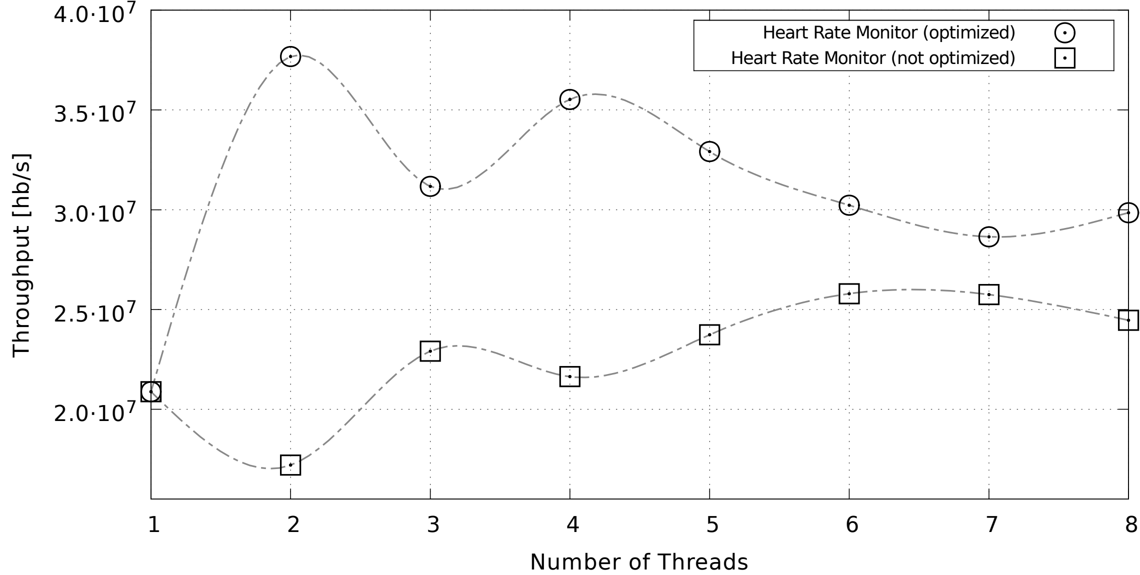


Figure 6.1: Comparison between the maximum heartbeats throughput achievable with HRM with and without the cache-alignment optimization. The measure is on the emission of 1 million heartbeats with 1 to 8 threads on platform A; the timer period is set to 100ms and the size of the window is 0 periods. The plots show the average of 1000 experiments; the maximum coefficient of variation (not reported in the graph) is $c_v = \frac{\sigma}{\mu} = 0.19$. Higher is better

platform A is dual-core). The non-optimized version, on the other hand, reaches the worst performance with two threads: this result is due to the high traffic generated on the bus to update the falsely-shared counters in the two private caches; the effect is emphasized by the fact that the two threads run on one core each and there is no serialization, thus increasing the contention on the falsely shared memory areas.

The second experiment regarding the monitoring overhead has been conducted on platform B (which features a quad-core processor) and, beyond characterizing the overhead of HRM on a different microarchitecture, it also allows a comparison with Application Heartbeats (see 3.1.3), a state of the art open source [24] monitor which leverages the same basic concepts at the base of HRM. To allow this comparison, tachycardia has been instrumented with the library offered by Application Heartbeats (using the shared memory option for communication, instead of the slower one based on files), and the same test of emitting 1 million

heartbeats as fast as possible has been run with both HRM and Application Heartbeats on platform B. Figure 6.2 shows the results of these experiments.

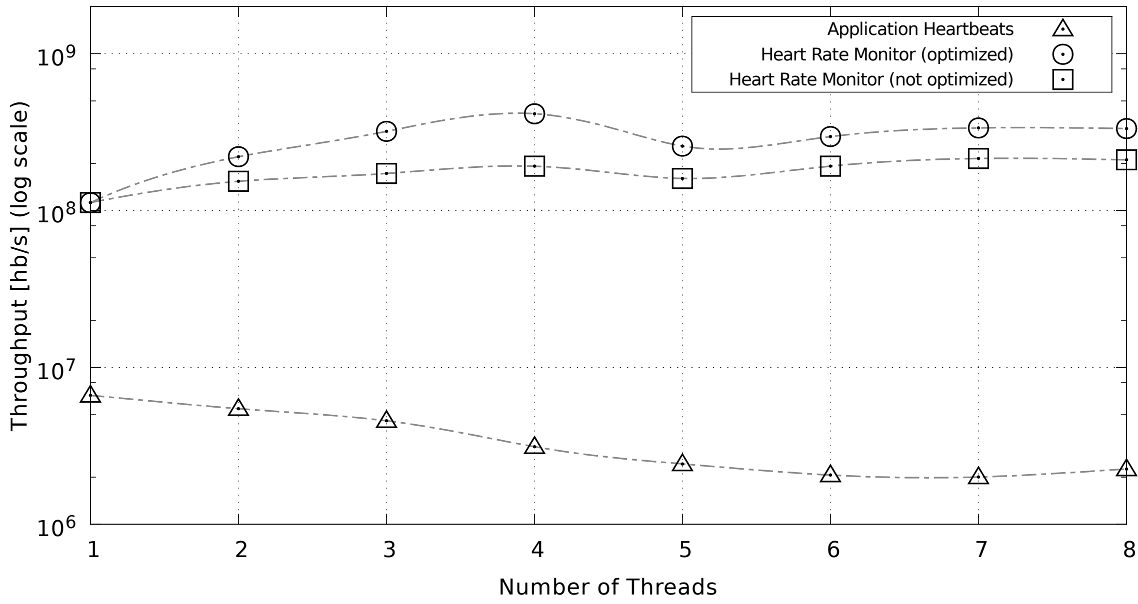


Figure 6.2: Comparison between the maximum heartbeats throughput achievable with HRM and Application Heartbeats. The measure is on the emission of 1 million heartbeats with 1 to 8 threads on platform B; the timer period is set to 100ms and the size of the window is 0 periods. The plots show the average of 1000 experiments; the maximum coefficient of variation (not reported in the graph) is $c_v = \frac{\sigma}{\mu} = 0.16$. Higher is better

The Figure represents the throughput achieved with both the optimized and the non-optimized version of HRM and the throughput ensured by Application Heartbeats. As the graphs show, both versions of HRM are mush faster than Application Heartbeats, with a speedup from one to two orders of magnitude times faster. Moreover, Application Heartbeats does not scale at all with respect to the number of threads, as it yields maximum throughput with only one thread.

The results of these two experiments demonstrate that the design and implementation of HRM achieved the efficiency goals for which it was engineered. Moreover, the comparison with Application Heartbeats demonstrates that HRM, beyond providing more functionalities (e.g., support for multi-processed applications), also guarantees a much lower overhead on the

monitored applications.

6.3.2 Characterization of a Real Workload

The third experiment regarding HRM is its use to characterize the execution of a run of x264 encoding the reference video. HRM is capable of exporting both the global heart rate of a group (i.e., the average heartbeats emitted per second from the beginning of the execution up to the current instant) and the window heart rate, which is measured only a time window and considers only the latest emitted heartbeats (up to the depth of the window), discarding the previous history. The depth of the time window can be set at runtime both by the applications and by the users; using a window of different lengths means considering a different span of time and thus providing different measurements. This experiment, and all the subsequent ones, has been realized on platform C, which features a quad-core processor. For this test, a 4-threaded instance of x264 has been run with the *superfast* preset and its execution has been recorded by using five different window sizes: from 1s to 10s. Figure 6.3 shows these results. As the Figure shows, increasing the window size smooths the recorded heart rate, as it averages the emitted heartbeats over a longer time span. In particular, the measurements done with the window size do 1s are quite irregular, as it records very short time trends during the execution. At the opposite side, the last plot is very smooth, and it tends to overlap with the global heart rate measure. Among the five runs, the one in the middle, i.e., the one with a window size of 5s seems the most balanced one, capturing the fluctuations but avoiding too quick variations. After this quick analysis, a window size of 5s has been chosen for the remaining experiments. Clearly, this parameter depends on the application and on the data; the choice based on this analysis is justified by the fact that all the remaining tests are realized with the same application and on the same data.

Another interesting observation that can be made on Figure 6.3 is that the first part of execution proceeds at a faster pace (with the window heart rate being above the global heart rate), then the two lines are nearer and, in the last part, the heart rate decreases a little. These fluctuations are due to the video being encoded, which clearly requires more intensive computations in the last part. Since also the remaining tests use the same video, these trends can be found also in the following results regarding PAFS.

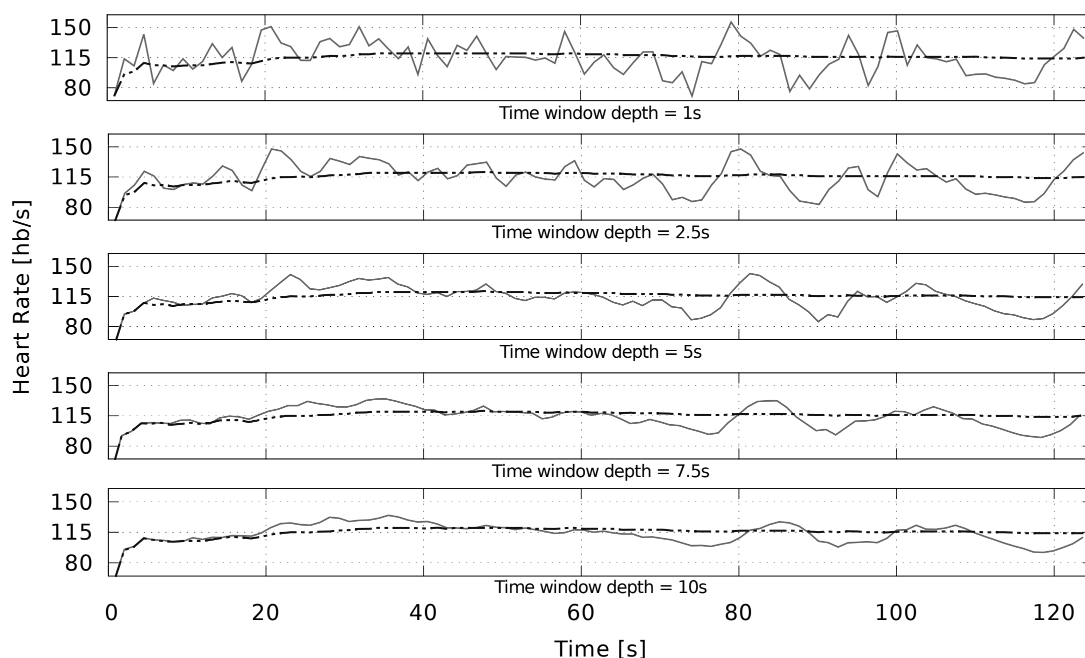


Figure 6.3: Characterization of the execution of a real workload with HRM using different durations for the time window. The five plots show five runs of x264 encoding the reference video with preset *superfast* using 4 threads. The timer period is set to 100ms and, for each run, the size of the time window has been changed, from 10 periods (i.e., 1s) to 10s. The black dotted line shows the global heart rate, while the gray line plots the window heart rate

6.4 Adaptive Scheduling Capabilities

The evaluation of PAFS has been realized using the same workload characterized with HRM alone in Section 6.3.2. In that experiment, the default Linux scheduler (i.e., the CFS) was used, without the patch enabling PAFS. The experiments illustrated in the present Section compare the behavior of the CFS with the behavior achievable with PAFS by setting performance goals for the monitored applications; all these experiments have bee realized on platform C, which is equipped with a quad-core microprocessor, and by using the window heart rate on the last 5 seconds as the current heart rate considered by PAFS (see Section 6.3.2 for the motivation of this choice). The idea is to evaluate how PAFS can prove useful in some likely use cases for PAFS. The tests have been engineered to characterize the system by both showing its strengths and finding its weaknesses, which will be targeted in future works.

6.4.1 Contrasting External Load

The first use case for PAFS is the execution of one instrumented application on a highly loaded system. In this scenario, the proprietary of the application may want to boost its performance and, to do so, he/she could use PAFS to set a goal heart rate. Clearly, if different users share the same system, each one could possibly be willing to maximize the performance of his/her own applications, hence penalizing the others. The other users, however can always react by setting higher goals for their applications, and so forward. In such a setup, the system's computing resources will not be enough to reach all the goals of the applications. The final result, will be that all the monitored applications will be equally advantaged over the non monitored ones, leading to fairness among the evil-behaving users. This is a nice side-effect of how PAFS is plugged in into the CFS, preserving the baseline property of fairness.

Under the hypothesis that only one application is HRM enabled in the system, it is possible to evaluate the functionality of PAFS in coping with the need of boosting its performance against high system load by setting different goals and verifying if PAFS is able to let the application achieve them. To do so, a run of x264 encoding the reference video with the *ultrafast* preset (which is faster that *veryfast*, and has been chosen to reduce the duration of the test) has been recorded both with and without system load. The background system load has been obtained by executing four (as the number of available cores) instances of the `burnP6` stress test. Figure 6.4 shows the results of this measurements This first measure was taken without setting any goal (i.e., setting both minimum and maximum heart rate at 0): in this situation, PAFS behaves just as the plain CFS; the results of this runs are superimposed in Figure 6.4 (in this case, despite being reported on the same plot to allow a comparison, the two encoders did not run concurrently). As the Figure shows, the execution with high system load is far slower that without the additional load, taking less than 100 seconds - with a global heart rate of about $150\frac{hb}{s}$ - versus more than 7 minutes - with a global heart rate of just $31\frac{hb}{s}$. The aim of the experiment is being able, via setting heart rate goals higher than $31\frac{hb}{s}$, to boost the performance of the application running on an overloaded system. to verify this capability, the possibility for the user to set the desired heart rate through the `/proc/$PID/task/$TID/hrm_goal` pseudo-file has been exploited. More in detail, an initial goal of $[25 : 35]\frac{hb}{s}$ (i.e., a minimum heart rate of $25\frac{hb}{s}$ and a maximum heart rate of $35\frac{hb}{s}$)

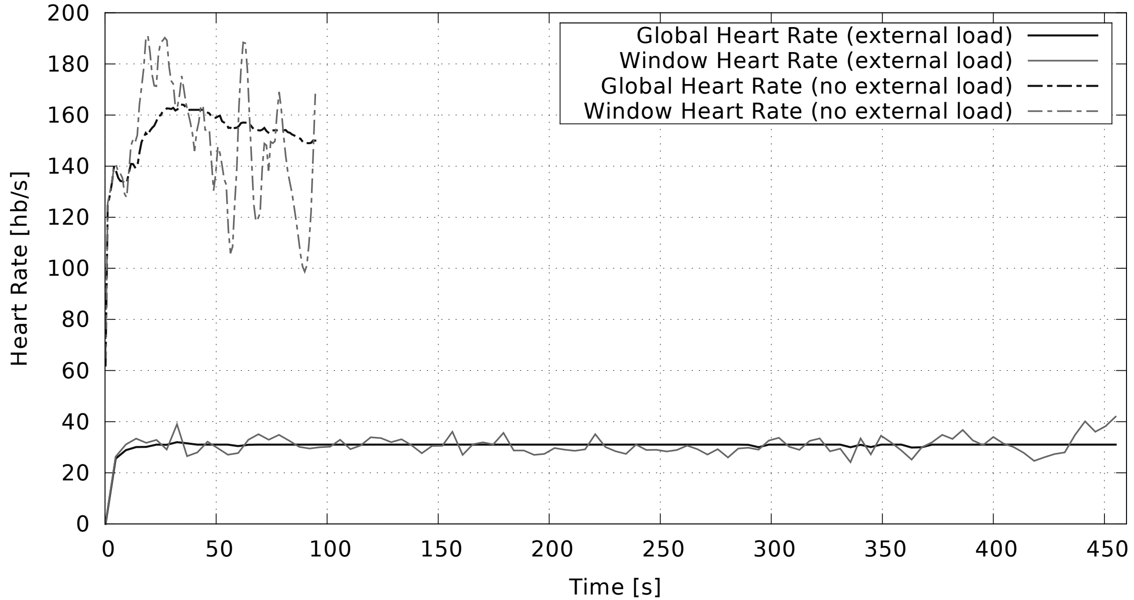


Figure 6.4: Comparison of the execution traces of the global and window heart rates of a video encoder with and without high system load. Both traces regard a 4-threaded x264 encoder with preset *ultrafast*; the one proceeding faster has been recorded with no additional system load, the other had four `burnP6` instances running simultaneously

has been set and kept for 40 seconds. Then, the goal was changed to $[45 : 55]\frac{hb}{s}$ for 40 more seconds; the same thing has been done by changing the goal to $[65 : 75]\frac{hb}{s}$ and, finally, to $[85 : 95]\frac{hb}{s}$. The four instances of `burnP6` have been kept running for the whole duration of the test. Figure 6.5 represents the results of this experiment.

In the Figure, the shaded areas represent the heart rate goals changed during the execution as described above. The first desired range captures the current performance of the application, and thus PAFS does not act differently from the CFS, but for smoothing its performance on the short term when it would have gone over the upper bound (which, in any case, is considered as a *soft* bound - see Section 4.2.5). When the desired heart rate is changed, PAFS is able to speed up the encoder, being able to keep the window heart rate over the minimum of $45\frac{hb}{s}$ for almost all the 40 seconds. At the next change of the goal, the action of PAFS becomes weaker, being able to meet the minimum required QoS only partially during the execution. The last goal is evidently too high for the system to achieve it, and PAFS just

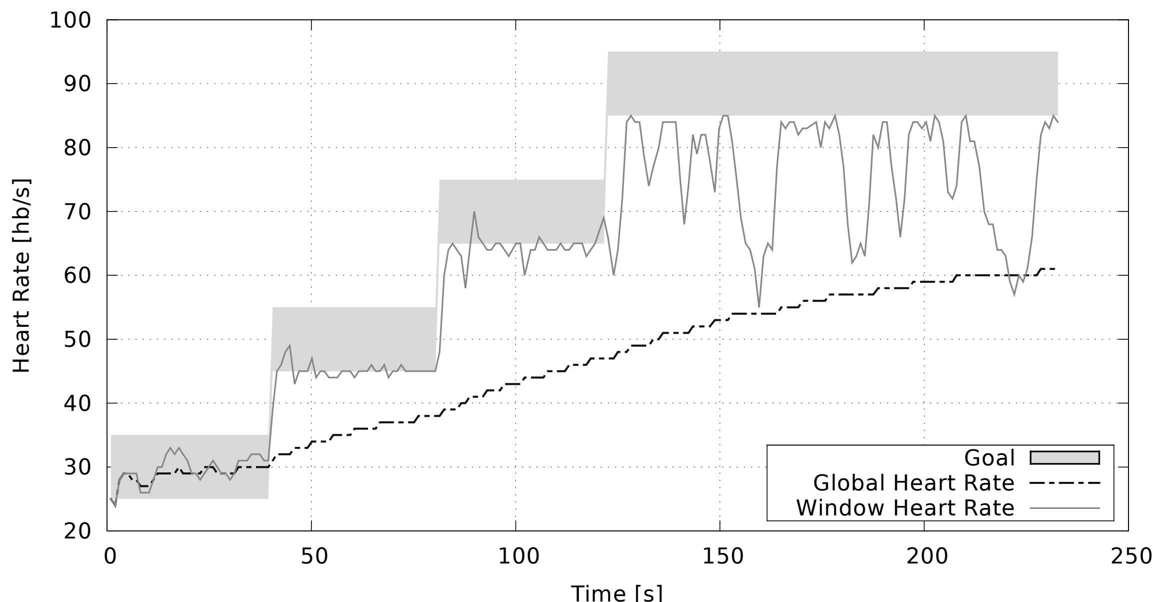


Figure 6.5: Execution trace of the global and window heart rates of a video encoder with performance goals under high system load. The application is a 4-threaded x264 encoder with preset *ultrafast*; the shaded areas represent the goal for the application

advantages the encoder as much as possible, but without reaching the desired heart rate. Notice that the total execution time is now about 230s, with a global heart rate over $60\frac{hb}{s}$: this represents a speedup of 2× with respect to the situation with no goals set (which is represented in Figure 6.4). Even when the desired goal could be reached (i.e., from second 40 to second 120), PAFS was not able to keep the window heart rate well within the desired range, but is just able to keep it a little above the minimum. This limitation is due to the simplicity of the heuristic implemented for this thesis and could probably be overcome with a smarter policy. Even with this simple decision mechanism, however, PAFS proves to be able to boost the performance of an instrumented application towards reachable goals.

6.4.2 Performance Separation

The experiment proposed in Section 6.4.1 considered the scenario where only one instrumented application is running on a heavily loaded system. A different use case is when two

instrumented applications with similar heart rates (and, hence, similar execution times) are running concurrently. In this situation, it could be the case that the user desires one of the two applications to run faster, sacrificing the performance of the other to get a shorter execution time. This situation has been modeled by using two 4-threaded instances of x264 running concurrently with the preset *superfast* to encode two copies of the reference video (in this case, no additional load is applied). The runtime behavior of the two applications (i.e., Group 1 and Group 2) in this scenario when no performance goal is set is represented in Figure 6.6. As expected, the Figure shows that both the global and the window heart rates of the two

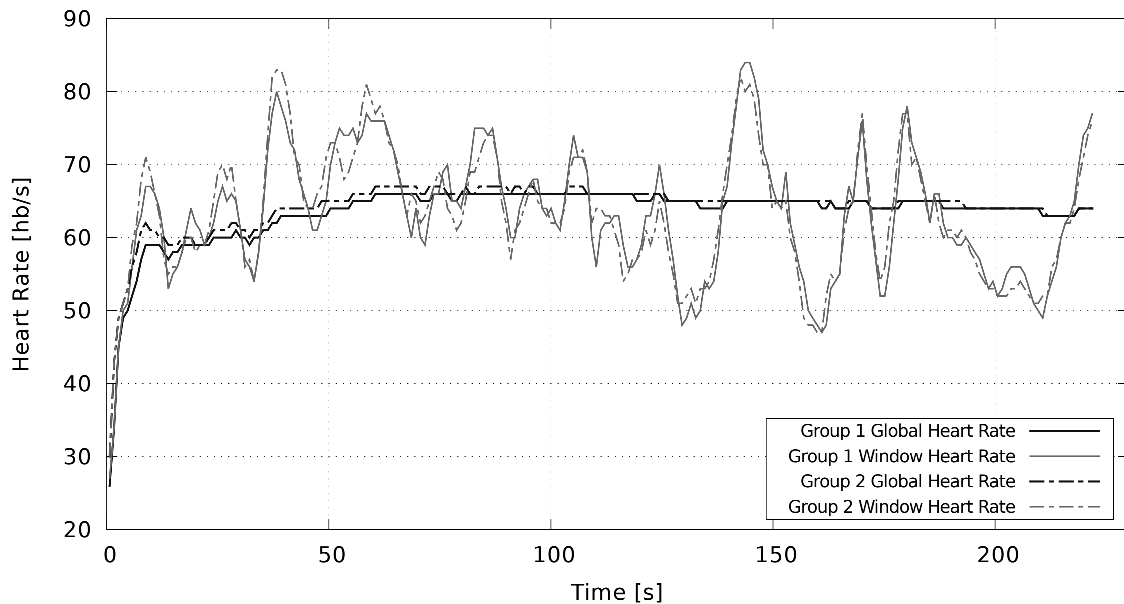


Figure 6.6: Window and global heart rates during the execution of two applications performing the same job with no performance goals. The two groups are two 4-threaded x264 encoders working on the reference video with the *superfast* preset

encoders are almost overlapping: when no goal is set, PAFS behaves just like the CFS and applies complete fairness among the tasks in execution. The purpose of this test is verifying if PAFS is able to boost one of the two applications, while slowing down the other. To verify this capability, the performance goal has been set to $[5:40]\frac{hb}{s}$ for Group 1 and to $[70:100]\frac{hb}{s}$ for Group 2; note that the global and window heart rates are out of both of these bounds

for most of the time when no goal is set (see Figure 6.6). Figure 6.7 represents the results of setting the two different goals for the two homogeneous groups.

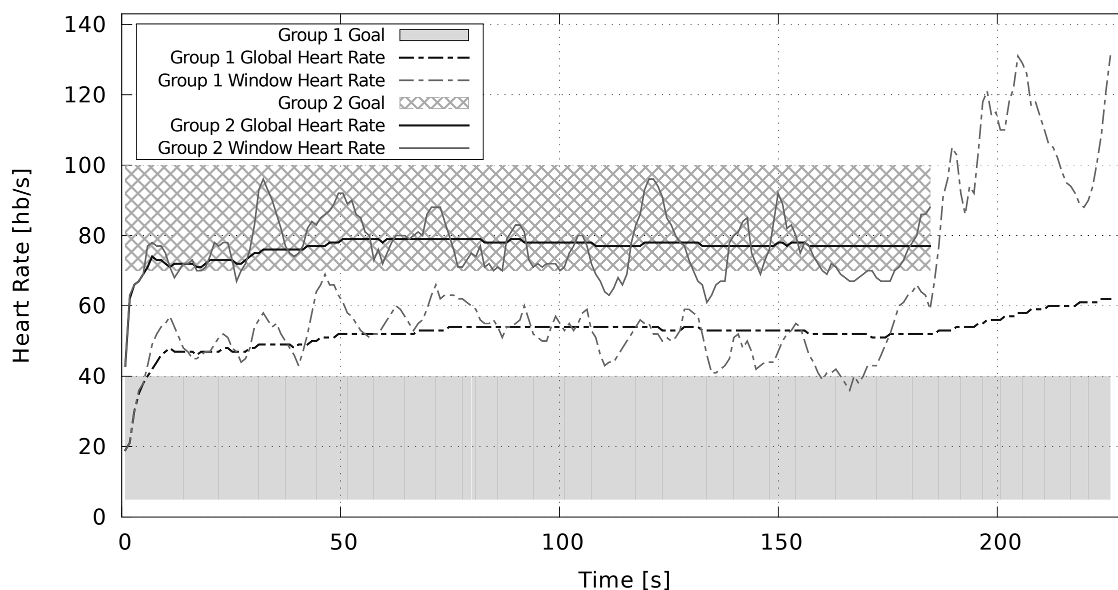


Figure 6.7: Window and global heart rates of two applications performing the same job with different performance goals. The two groups are two 4-threaded x264 encoders working on the reference video with the *superfast* preset; the goal for Group 1 is set to $[5:40]\frac{hb}{s}$, while the goal for Group 2 is at $[70:100]\frac{hb}{s}$

As the traces in the Figure show, the system was able to speed up the execution of the encoder instrumented in Group 2, keeping both its global and window heart rate above its minimum bound for most of the execution time. The other encoder, instrumented in Group 1 overperforms its upper bound for almost all the execution time: in fact, it would have been unnecessary to slow it down further, as all the monitored applications were running faster than their minimum desired heart rate. also notice that, when the execution of Group 2 terminates, Group 1 receives a huge speed up: this is due to the fact that it is now the only application in execution on the system and. Thus, with no contention for the computing resources (the encoder has 4 threads, as many as the processor's cores, and its being executed alone), the action of PAFS on its vruntime has no sensible effect on the performance of the application. Again, this situation is due to how the implemented heuristic works and to the interpretation

given to the upper bound. In some situations, it could be useful to use the maximum heart rate as a performance cap: further studies in this direction are left for future works. Another thing to notice is that speeding up one of the two encoders did not really affect the overall execution time, as both encoders terminate after about 230s, just as when no goals was set.

6.4.3 Performance Inversion

The last use case proposed for PAFS is the situation where there are two concurrent applications with different performance: one of the two is faster (i.e., it has a faster execution time) than the other. In this situation, it could be sometimes desirable to speed up the slower application to have it finish before the other, for instance because the result it provides is currently more urgent than the other. This context has been modeled by having two 4-threaded x264 encoders working on two copies of the reference video with different presets (i.e., *superfast* and *ultrafast*). As usual, the two applications have been first concurrently run with no goals set, to evaluate the base case for the experiment. The plot in Figure 6.8 represents this measure. As

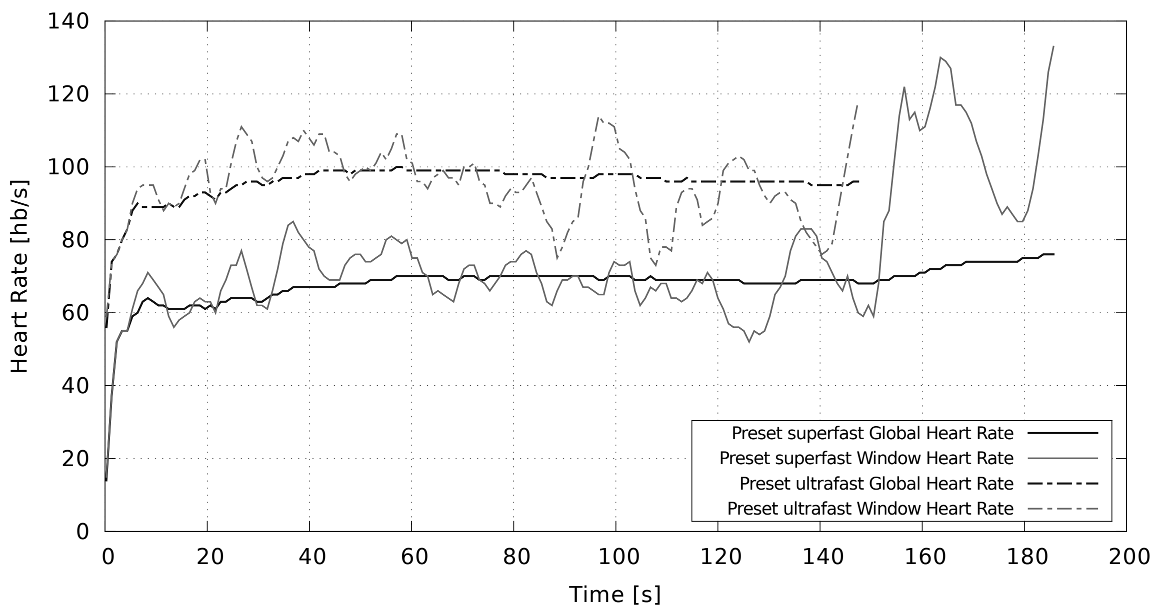


Figure 6.8: Window and global heart rates during the execution of two video encoders running with different presets and no performance goals. The two groups are two 4-threaded x264 encoders working on the reference video, respectively with the - slower - *superfast* and the - faster -*ultrafast* preset

shown in the Figure, the preset *ultrafast* is faster, yielding a global heart rate of almost $100\frac{hb}{s}$, against a global heart rate of about $65\frac{hb}{s}$ of the group corresponding to the encoder working with *superfast* preset. Similarly as before, the experiment consists in modifying the runtime behavior by setting a lower goal for the faster group and a higher goal for the slower group. Hence, the minimum heart rate for the *superfast* encoder is set to $80\frac{hb}{s}$, and its maximum heart rate is set to $115\frac{hb}{s}$; the performance goal for the *ultrafast* encoder is set to $[30:65]\frac{hb}{s}$. Figure 6.9 plots the heart rates measured with these goals set.

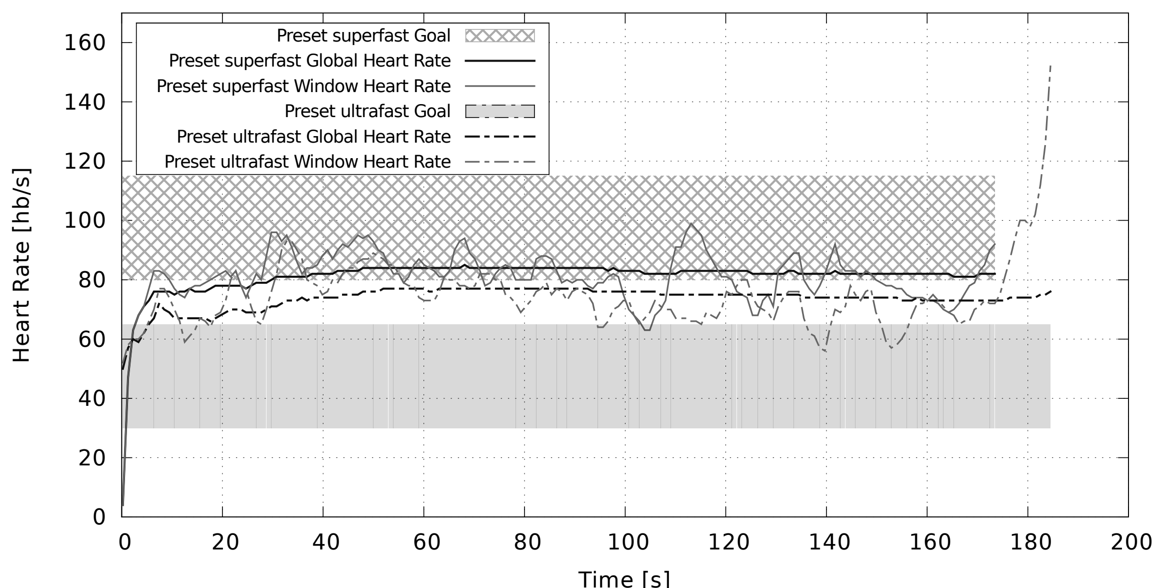


Figure 6.9: Window and global heart rates during the execution of two video encoders running with different presets and inverse performance goals. The two groups are two 4-threaded x264 encoders working on the reference video, respectively with the - slower - *superfast* and the - faster -*ultrafast* preset. The goal for the *superfast* encoder is set to $[80:115]\frac{hb}{s}$, while the *ultrafast* encoder's goal is at $[30:65]\frac{hb}{s}$

As the Figure shows, PAFS has been able to speed up the slower encoder just a little above its minimum heart rate, while the faster encoder outperforms its upper bound for almost all the execution time. The speedup of the *superfast* encoder, however, is sufficient for letting it terminate before the *ultrafast* encoder, capsizing the situation when no goals are set. Just as before, when the *superfast* encoder terminates, the performance of the *ultrafast* springs up, as it is the only application in execution. Again, notice that the overall execution time needed

for terminating the two jobs is approximatively equal to the time needed when no goals are set.

Since the strength of the autonomic action of PAFS highly depends on the amount of contention for the computing resources, it is possible to increase this quantity to see if the resulting effect is enhanced. In all the experiments presented up to this point, the encoders were run with 4 working threads, which is the optimal choice for a quad-core processor, as the one present on platform C, which is used in all these tests. An additional experiment has been performed with two 8-threaded encoders, one with the *superfast* and one with the *ultrafast* preset (just as in the previous experiment). As expected, adding more threads does not increase the performance of the encoders when no goals are set (the graph showing this situation is omitted here, since it is very similar to Figure 6.8). When it comes to setting performance goals, however, the increased contention for processor time enhances the action of PAFS, as shown in Figure 6.10.

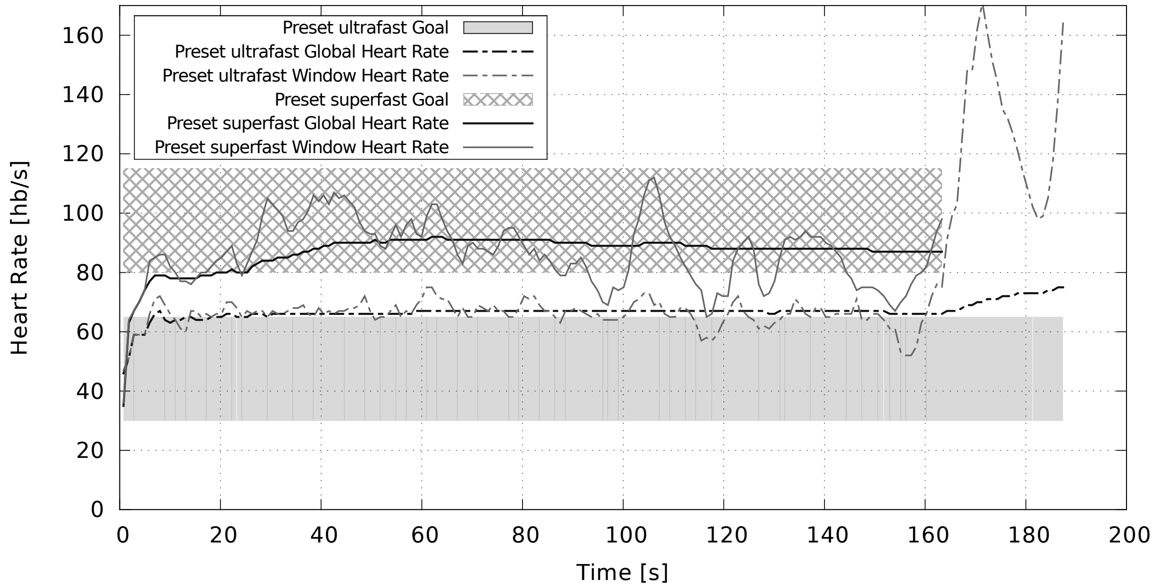


Figure 6.10: Window and global heart rates during the execution of two video encoders running with different presets and inverse performance goals. The two groups are two 8-threaded x264 encoders working on the reference video, respectively with the - slower - *superfast* and the - faster -*ultrafast* preset. The goal for the *superfast* encoder is set to $[80 : 115]\frac{hb}{s}$, while the *ultrafast* encoder's goal is at $[30 : 65]\frac{hb}{s}$

As shown in the Figure, this time PAFS has given further speedup to the *superfast* encoder, further slowing down the *ultrafast* one with respect to the experiment with 4-threaded encoders represented in Figure 6.9. Again, the performance inversion does not sensibly affect the overall execution time.

6.5 Summary

Within this Chapter, the results gathered through a variety of experiments performed to evaluate the proposed system are illustrated and analyzed. HRM proves to have achieved its goals of being lightweight and it outperforms a state of the art software monitor based on the same basic ideas. Moreover, the cache-alignment optimization allows the monitor to scale well with respect to the number of threads on multi-core processors, and the adjustable window size allows it to catch shorter or longer term trends, as needed by the monitored application. The evaluation of PAFS demonstrates that, even with the simple heuristic used as the decision mechanism, it is able to add the possibility of specifying goals in terms of QoS to a best effort scheduling approach. In fact, PAFS is often able to let the monitored groups achieve their goals when these are reachable with the available computing resources. To do so, however, there must be enough contention over the computing resources, otherwise the strength of the adaptation policy weakens, up to having no sensible effect. The results also highlight the limits of the current implementation of PAFS, which is enough to demonstrate the validity of the approach, but it leaves room for improvements (mainly in terms of a smarter decision mechanism), showing an interesting direction for future works on AcOS. In the next Chapter, some more detailed conclusions on the work described in this thesis are proposed and some possible directions for future works are suggested.

Chapter 7

Conclusions

The work described in this thesis provides both a theoretical formalization and the design and implementation of the enabling technologies to enhance commodity operating systems with autonomic capability of self-management in a goal-oriented scenario. After an introduction, which serves to define some basic concepts, the study of the state of the art and related works provides an overview of how existing projects and systems tackle the problems addressed by this thesis. These problems concern the ever increasing complexity of computing system and the overburden this complexity imposes on applications developers. The proposed approach to tackle these issues is based on the ideas coming from the autonomic computing research community and it includes proposals of renovation within all the layers of a computing system (i.e., hardware architecture, operating system, and applications). These innovations are based on the concept of a feedback self-management loop composed of the basic operations of *observation*, *decision*, and *action*. In particular, the OS has been identified as the most urgent portion of the system to be enhanced with autonomic capabilities; hence, this thesis is focused on the creation of the enabling technologies to build an autonomic operating system through the enhancement of commodity OSes with an autonomic layer. Within this context, a novel software monitor (i.e., HRM) and an adaptation policiy (i.e., PAFS) have been designed and implemented over Linux to prove the validity of the approach. This Chapter has the role of proposing some additional conclusive remarks (in Section 7.1) and of giving some cues for future works (in Section 7.2).

7.1 Concluding Remarks

As the results (analyzed in Chapter 6) of the experimental evaluation of the proposed system show, HRM and PAFS prove to be interesting enabling technologies to augment, at the OS-level, the self-management ability of computing systems. These first autonomic components deployed onto a commodity operating system prove the realizability of the approach and open the scene for future AcOS developments.

HRM is based on a sound design and a smart implementation, which result in outstanding efficiency (i.e., low overhead) with respect to a state of the art solution. Moreover, it is capable of supporting a wide range of parallel applications (supporting both multi-threaded and multi-processed groups), and it provides a flexible and general purpose means of instrumenting the applications to provide runtime information (in the form of an heart rate) to the autonomic layer. The provided statistics are pretty accurate (under the hypothesis of a sound instrumentation yielding a reasonable heart rate) and the availability of both a global heart rate and a window heart rate (computed over a runtime-tunable time span) allow to capture both long and short time trends.

A possible use of HRM is to measure the performance of CPU-bound applications characterized by a *hotspot*; this is the role the monitor covers in the performance aware adaptive scheduling control loop realized by PAFS, an adaptation policy built on top of Linux's CFS. PAFS, despite being currently based on a simple heuristic as the decision mechanism, is able to prove its usefulness in some use cases based on a real workload. In particular, by exploiting the possibility of specifying performance goals for the monitored applications, it is possible to drive their runtime behavior to satisfy the desired QoS (provided that this is possible with the available resources). This capability realizes the novel idea of performance aware fairness, introducing the possibility of expressing QoS requirements within a best effort context.

The experimental evaluation, however, also highlights some shortcomings of the implemented system, which are mainly related with the lack of a smart decision mechanism; these weaknesses indicate an interesting way for future works devoted to the creation of more elaborated decision engines to be used within the autonomic operating system. Some cues for future works in this and other directions are provided in the next Section.

7.2 Future works

As already highlighted, the experimental results showed the limitations of the simple decision mechanism used for PAFS. This heuristic is enough to prove the validity of the approach, but probably a more elaborated policy would better serve a production system. This observation indicates the first interesting area for future developments focused on the creation of smarted decision mechanisms for PAFS or different adaptation policies. Within this context, a viable possibility (which is already in phase of development within the CHANGE research group) is to exploit ideas from control theory to create decision engines based on feedback controllers with the capability of performing online identification of their parameters. Such controllers, are able to exploit knowledge about the past autonomic actions (which is not done by the heuristic currently used in PAFS) to capture trends in the evolution of the system and use this information to apply better corrective actions. Another interesting area for future works regarding the decision phase is the study of machine learning and artificial intelligence techniques to further improve the adaptation policies and to create the adaptation manager, which will have the role of orchestrating the different adaptation policies towards global system goals.

Lots of space for future works also lies in the creation of more autonomic components, which can be built over the base provided by the work presented in this thesis, to extend the capability of AcOS of building self-management and goal-orientation into computing systems. Possible ideas for building more autonomic capabilities in AcOS (some of which are already being explored within the CHANGE research group) include working on thermal or power consumption awareness by acting on frequency and voltage scaling, improving the locking mechanisms to reduce contention, and researching adaptivity in memory management.

Future works more specific to the proposed implementation could consider the following ideas:

- Concerning the use of the minimum and maximum heart rates as a performance goal, a different interpretation of the upper bound on the heart rate could be given, for instance, as a limit to the power consumption determined by the application. This should be possible by determining a relation between the heart rate and the power consumption due to that application. With this interpretation, also the upper bound would be *hard*

and the system should always strive to keep the heart rate of the applications below that limit. The work on determining a meaningful relationship between the heart rate and the power consumption is not trivial (especially on multicore or heterogeneous architectures) and research in this direction is being carried on within the CHANGE group. This interpretation is different from the *soft* one used by PAFS and it could come useful to different adaptation policies focusing, for isntance, on frequency and voltage scaling.

- PAFS is plugged into the CFS by affecting the update of the vruntime; future works may focus on trying to use different (or more) points of application for the autonomic action, for instance affecting the duration of the dynamic tics used in the CFS, or the weights used to compute the vruntime.

- Another possible work may be directed towards extending PAFS with the ability to act on the *core allocation* of the monitored tasks (which is currently left to the standard behavior defined by the CFS). Works in this direction could give to the control loop more power on the computing resources assignment, allowing a stronger action on the system status.

- The current implementation of the proposed system is based on Linux. Porting the codebase of HRM and PAFS to new releases of the kernel will not be a complex work, as long as no major changes are performed to the mainline implementation of the portions of the kernel code that were modified. However, it could be interesting to port HRM to different platforms, to allow the use of the same performance monitor on devices not supported by Linux. Clearly, porting HRM to a platform completely different from Linux would probably require re-engineering part of the monitor; a port to a similar monolithic kernel, on the other hand, is an easier task (in fact, a HRM port to the FreeBSD kernel has already been realized).

Another very interesting area for broader future works, which the author will probably start exploring in first person in the near future, is the creation of a standalone autonomic operating system. This means creating a new kernel (or using a very thin microkernel as a base), to build AcOS from the roots, instead of just creating an autonomic layer over a

commodity operating system such as Linux. Clearly, a project of this type requires a longer time to come to interesting results (which is the main reason behind the choice of Linux in the first place). The experience gained through this and other works, however, could be invested in the creation of an operating system built from its roots with autonomic management in mind.

In conclusion, this thesis creates a base for AcOS, proposing a Linux-based implementation of an operating system able to build performance awareness into the process scheduling process. The results showed in the experimental evaluation of the system demonstrate that the approach is valid and that it is suitable to serve as a basis for future works, being just incremental improvements, or complete revolutions.

Bibliography

[1] Josh Aas. *Understanding the Linux 2.6.8.1 CPU Scheduler.* Tech. rep. Silicon Graphics, Inc. (SGI), 2005 (cit. on pp. 17, 21, 44).

[2] Anant Agarwal and Markus Levy. "The kill rule for multicore". In: *Proceedings of the 44th annual Design Automation Conference.* DAC '07. San Diego, California: ACM, 2007, pp. 750–753. ISBN: 978-1-59593-627-1. DOI: http://doi.acm.org/10.1145/1278480.1278668. URL: http://doi.acm.org/10.1145/1278480.1278668 (cit. on p. 24).

[3] J. Appavoo et al. *Scheduling in k42.* Tech. rep. White paper. [Online]. Available: http://www.research.ibm.com/K42/white-papers/Scheduling.pdf. IBM T. J. Watson Research Center, 2002 (cit. on pp. 34, 35).

[4] Jonathan Appavoo et al. "An infrastructure for multiprocessor run-time adaptation". In: *Proceedings of the first workshop on Self-healing systems.* WOSS '02. Charleston, South Carolina: ACM, 2002, pp. 3–8. ISBN: 1-58113-609-9. DOI: http://doi.acm.org/10.1145/582128.582130. URL: http://doi.acm.org/10.1145/582128.582130 (cit. on p. 33).

[5] Jonathan Appavoo et al. *Enabling autonomic system software with hot-swapping.* Tech. rep. [Online]. Available: http://www.research.ibm.com/K42/white-papers/MemoryMgmt.pdf. IBM T. J. Watson Research Center, 2003 (cit. on p. 33).

[6] Jonathan Appavoo et al. *K42's Performance Monitoring and Tracing.* Tech. rep. White paper. [Online]. Available: http://www.research.ibm.com/K42/white-papers/MemoryMgmt.pdf. IBM T. J. Watson Research Center, 2002 (cit. on pp. 41, 42).

[7] Marc Auslander et al. *Enabling Scalable Performance for General Purpose Workloads on Shared Memory Multiprocessors.* Tech. rep. Technical Report RC22863, [Online] Available: http://domino.research.ibm.com/comm/research.nsf/pages/r.os.innovation.html/\protect\T1\textdollarFILE/scalability-techrep.pdf. International Business Machines (IBM), 2003 (cit. on p. 32).

[8] Marc Auslander et al. *K42 Overview*. Tech. rep. White paper. [Online]. Available: `http://www.research.ibm.com/K42/white-papers/Overview.pdf`. IBM T. J. Watson Research Center, 2002 (cit. on pp. 9, 32, 33).

[9] Various authors. *Big Buck Bunny video download website*. [Online] Available: `http://www.bigbuckbunny.org/index.php/download/`. 2011 (cit. on p. 105).

[10] Various Authors. *CHANGE Research Group website*. [Online] Available: `http://www.changegrp.org/`. 2011 (cit. on pp. 55, 56, 80).

[11] Various Authors. *Debian web site*. [Online] Available: `http://www.debian.org/`. 2011 (cit. on p. 105).

[12] S. Baskiyar and N. Meghanathan. "A Survey of Contemporary Real-time Operating Systems". In: *Networks, Parallel and Distributed Processing, and Applications (NPDPA)*. 2002 (cit. on p. 49).

[13] Andrew Baumann et al. "Providing dynamic update in an operating system". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 32–32. URL: `http://portal.acm.org/citation.cfm?id=1247360.1247392` (cit. on p. 33).

[14] Andrew Baumann et al. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629579`. URL: `http://dx.doi.org/10.1145/1629575.1629579` (cit. on pp. 9, 24, 27, 28).

[15] Silas Boyd-Wickizer et al. "An Analysis of Linux Scalability to Many Cores". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. Vancouver, Canada, 2010 (cit. on p. 24).

[16] Scott A. Brandt et al. "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes". In: *24th IEEE Real-Time Systems Symposium*. 2003, pp. 396–407 (cit. on pp. 16, 47, 48).

[17] Pascal Brisset et al. *ECLiPSe 3.5 - ECRC Common Logic Programming System - Extensions User Manual*. 1995 (cit. on p. 29).

[18] S. Browne et al. "A Portable Programming Interface for Performance Evaluation on Modern Processors". In: *International Journal of High Performance Computing Applications* 14.3 (2000), pp. 189–204 (cit. on p. 40).

[19] Calin Cascaval et al. "Performance and environment monitoring for continuous program optimization". In: *IBM Journal of Research and Development* 50.2–3 (2006), pp. 239–248 (cit. on p. 42).

[20] Guilin Chen et al. "An Adaptive Locality-Conscious Process Scheduler for Embedded Systems". In: *Real-Time and Embedded Technology and Applications Symposium, IEEE* 0 (2005), pp. 354–364. ISSN: 1080-1812. DOI: http://doi.ieeecomputersociety.org/10.1109/RTAS.2005.6 (cit. on p. 51).

[21] Julita Corbalan, Xavier Martorell, and Jesus Labarta. "Performance-Driven Processor Allocation". In: *IEEE Trans. Parallel Distrib. Syst.* 16 (7 2005), pp. 599–611. ISSN: 1045-9219. DOI: 10.1109/TPDS.2005.85. URL: http://dl.acm.org/citation.cfm?id=1070608.1070725 (cit. on p. 52).

[22] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN: 0596005903 (cit. on pp. 81, 83).

[23] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. McGraw-Hill Higher Education, 2009 (cit. on pp. 44, 46).

[24] MIT CSAIL. *Application Heartbeats Google Code repository.* [Online] Available: http://code.google.com/p/heartbeats/. 2011 (cit. on p. 109).

[25] Tommaso Cucinotta et al. "A robust mechanism for adaptive scheduling of multimedia applications". In: *ACM Transactions on Embedded Computing Systems*. 2009 (cit. on p. 47).

[26] Tommaso Cucinotta et al. "Self-tuning schedulers for legacy real-time applications". In: *Proceedings of the 5th European conference on Computer systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 55–68. ISBN: 978-1-60558-577-2. DOI: http://doi.acm.org/10.1145/1755913.1755921. URL: http://doi.acm.org/10.1145/1755913.1755921 (cit. on pp. 20, 52, 66).

[27] Tommaso Cucinotta et al. "The wizard of OS: a heartbeat for Legacy multimedia applications". In: *ESTImedia*. Ed. by Andy D. Pimentel and Naehyuck Chang. IEEE, 2009, pp. 70–79. ISBN: 978-1-4244-5170-8 (cit. on p. 66).

[28] N. Delgado, A.Q. Gates, and S. Roach. "A taxonomy and catalog of runtime software-fault monitoring tools". In: *Software Engineering, IEEE Transactions on* 30.12 (2004), pp. 859 –872. ISSN: 0098-5589 (cit. on pp. 11, 41).

[29] Petre Dini et al. "Internet, GRID, Self-Adaptability and Beyond: Are We Ready?" In: *Database and Expert Systems Applications, International Workshop on* 0 (2004), pp. 782–788. ISSN: 1529-4188. DOI: http://doi.ieeecomputersociety.org/10.1109/DEXA.2004.1333571 (cit. on p. 8).

[30] L. Dozio and P. Mantegazza. "Linux Real Time Application Interface (RTAI) in low cost high performance motion control". In: *Motion Control* (2003), pp. 27–28 (cit. on pp. 47, 49, 50).

[31] Jonathan Eastep et al. "Smartlocks: lock acquisition scheduling for self-aware synchronization". In: *Proceeding of the 7th international conference on Autonomic computing*. ICAC '10. Washington, DC, USA: ACM, 2010, pp. 215–224. ISBN: 978-1-4503-0074-2. DOI: http://doi.acm.org/10.1145/1809049.1809079. URL: http://doi.acm.org/10.1145/1809049.1809079 (cit. on p. 64).

[32] Manuel Fähndrich et al. "Language support for fast and reliable message-based communication in singularity OS". In: *SIGOPS OSR* 40.4 (2006), pp. 177–190. ISSN: 0163-5980. DOI: 10.1145/1218063.1217953. URL: http://dx.doi.org/10.1145/1218063.1217953 (cit. on p. 31).

[33] Ben Gamsa et al. "Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system". In: *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 87–100. ISBN: 1880446391. URL: http://portal.acm.org/citation.cfm?id=296814 (cit. on p. 32).

[34] Max Hailperin. *Changes in the Linux Scheduler as of 2.6.23*. [Online] Available: https://gustavus.edu/+max/os-book/updates/CFS.html. 2007 (cit. on p. 46).

[35] Bruce Hendrickson. "Emerging challenges and opportunities in parallel computing: the cretaceous redux?" In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. SPAA '09. Calgary, AB, Canada: ACM, 2009, pp. 130–130. ISBN: 978-1-60558-606-9. DOI: http://doi.acm.org/10.1145/1583991.1584029. URL: http://doi.acm.org/10.1145/1583991.1584029 (cit. on p. xviii).

[36] Benjamin Hindman et al. *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*. Tech. rep. UCB/EECS-2010-87. EECS Department, University of California, Berkeley, 2010. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-87.pdf (cit. on p. 9).

[37] Henry Hoffmann et al. "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments". In: *Proceeding of the 7th international conference on Autonomic computing*. ICAC '10. Washington, DC, USA: ACM, 2010, pp. 79–88. ISBN: 978-1-4503-0074-2. DOI: http://doi.acm.org/10.1145/1809049.1809065. URL: http://doi.acm.org/10.1145/1809049.1809065 (cit. on pp. 41–43, 62, 64).

[38] Henry Hoffmann et al. *SEEC: A Framework for Self-aware Computing*. Tech. rep. Massachussets Institute of Technology, 2010 (cit. on pp. 9, 10, 24, 32, 36, 37, 68).

[39] R. Hofmann et al. “Distributed performance monitoring: methods, tools, and applications”. In: *Parallel and Distributed Systems, IEEE Transactions on* 5.6 (1994), pp. 585 –598. ISSN: 1045-9219. DOI: `10.1109/71.285605` (cit. on p. 11).

[40] Paul Horn. *Autonomic computing: IBM’s perspective on the state of information technology.* [Online] Available: `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`. 2001 (cit. on pp. xviii, 2, 3).

[41] Markus C. Huebscher and Julie A. McCann. “A survey of autonomic computing-degrees, models, and applications”. In: *ACM Comput. Surv.* 40 (3 2008), 7:1–7:28. ISSN: 0360-0300. DOI: `http://doi.acm.org/10.1145/1380584.1380585`. URL: `http://doi.acm.org/10.1145/1380584.1380585` (cit. on pp. xix, 5, 6, 10–12, 54).

[42] Galen C. Hunt and James R. Larus. “Singularity: Rethinking the Software Stack”. In: *SIGOPS OSR* 41.2 (2007), pp. 37–49. ISSN: 0163-5980. DOI: `http://doi.acm.org/10.1145/1243418.1243424`. URL: `http://portal.acm.org/citation.cfm?id=1243424` (cit. on pp. 9, 30).

[43] Google inc. *Android web site.* [Online] Available: `http://www.android.com/`. 2011 (cit. on p. 104).

[44] Jasmina Jancic et al. “dproc - Extensible Run-Time Resource Monitoring for Cluster Applications”. In: *Proceedings of the International Conference on Computational Science-Part II.* ICCS ’02. London, UK, UK: Springer-Verlag, 2002, pp. 894–903. ISBN: 3-540-43593-X. URL: `http://dl.acm.org/citation.cfm?id=645458.655643` (cit. on p. 41).

[45] M. Tim Jones. *Inside the Linux 2.6 Completely Fair Scheduler.* [Online] Available: `http://public.dhe.ibm.com/software/dw/linux/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf`. 2009 (cit. on pp. 17, 21, 45).

[46] Mehdi Kargahi and Ali Movaghar. “A Method for Performance Analysis of Earliest-Deadline-First Scheduling Policy”. In: *J. Supercomput.* 37 (2 2006), pp. 197–222. ISSN: 0920-8542. DOI: `10.1007/s11227-006-5944-2`. URL: `http://portal.acm.org/citation.cfm?id=1145344.1145351` (cit. on p. 21).

[47] Jeffrey O. Kephart and David M. Chess. “The Vision of Autonomic Computing”. In: *Computer* 36 (1 2003), pp. 41–50. ISSN: 0018-9162. DOI: `http://dx.doi.org/10.1109/MC.2003.1160055`. URL: `http://dx.doi.org/10.1109/MC.2003.1160055` (cit. on pp. 3, 5).

[48] Avinesh Kumar. *Multiprocessing with the Completely Fair Scheduler.* [Online] Available: `http://download.boulder.ibm.com/ibmdl/pub/software/dw/linux/l-cfs/l-cfs-pdf.pdf`. 2008 (cit. on p. 45).

[49] Sungsoo Lim and Sung-Bae Cho. "Intelligent OS process scheduling using fuzzy inference with user models". In: *Proceedings of the 20th international conference on Industrial, engineering, and other applications of applied intelligent systems*. IEA/AIE'07. Kyoto, Japan: Springer-Verlag, 2007, pp. 725–734. ISBN: 978-3-540-73322-5. URL: http://portal.acm.org/citation.cfm?id=1769938.1770030 (cit. on p. 51).

[50] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20 (1 1973), pp. 46–61. ISSN: 0004-5411. DOI: http://doi.acm.org/10.1145/321738.321743. URL: http://doi.acm.org/10.1145/321738.321743 (cit. on p. 20).

[51] L. Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, 1998. ISBN: 0136566952 (cit. on p. 102).

[52] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. ISBN: 0672329468, 9780672329463 (cit. on pp. 14, 17, 19, 44–47, 75, 82, 92, 97, 98).

[53] K. Modzelewski et al. "A Unified Operating System for Clouds and Manycore: fos". In: *Proceedings of the first workshop on Computer Architecture and Operating System co-design* (2010). URL: http://mit.dspace.org/handle/1721.1/49844 (cit. on p. 25).

[54] Jason Nieh and Monica S. Lam. "A SMART scheduler for multimedia applications". In: *ACM Trans. Comput. Syst.* 21 (2 2003), pp. 117–163. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/762483.762484. URL: http://doi.acm.org/10.1145/762483.762484 (cit. on pp. 46, 48).

[55] Jason Nieh and Monica S. Lam. "The design, implementation and evaluation of SMART: a scheduler for multimedia applications". In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*. SOSP '97. Saint Malo, France: ACM, 1997, pp. 184–197. ISBN: 0-89791-916-5. DOI: http://doi.acm.org/10.1145/268998.266677. URL: http://doi.acm.org/10.1145/268998.266677 (cit. on pp. 48, 49).

[56] Edmund B. Nightingale et al. "Helios: heterogeneous multiprocessing with satellite kernels". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 221–234. ISBN: 978-1-60558-752-3. DOI: http://doi.acm.org/10.1145/1629575.1629597. URL: http://doi.acm.org/10.1145/1629575.1629597 (cit. on pp. 24, 29, 30).

[57] Videolan organization. *x264 website*. [Online] Available: http://www.videolan.org/developers/x264.html. 2011 (cit. on p. 105).

[58] Simon Peter et al. "Design principles for end-to-end multicore schedulers". In: *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. HotPar'10. Berkeley, CA: USENIX Association, 2010, pp. 10–10. URL: http://portal.acm.org/citation.cfm?id=1863086.1863096 (cit. on p. 28).

[59] Giovanni Racciu and Paolo Mantegazza. *RTAI 3.4 User Manual rev. 0.3*. 2006. URL: www.rtai.org (cit. on pp. 49, 50).

[60] Robert Redelmeier. *CPU burn website*. [Online] Available: http://pages.sbcglobal.net/redelm/. 2011 (cit. on p. 106).

[61] IBM research. *An Architectural Blueprint for Autonomic Computing*. [Online] Available: http://www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf. June 2006 (cit. on p. 32).

[62] Mazeiar Salehie and Ladan Tahvildari. "Self-adaptive software: Landscape and research challenges". In: *ACM Trans. Auton. Adapt. Syst.* 4 (2 2009), 14:1–14:42. ISSN: 1556-4665. DOI: http://doi.acm.org/10.1145/1516533.1516538. URL: http://doi.acm.org/10.1145/1516533.1516538 (cit. on pp. 3, 5).

[63] Marco Domenico Santambrogio. "A scheduling problem with conditional jobs solved by cutting planes and integer linear programming". 2007 (cit. on pp. 12, 13).

[64] Martin Schulz et al. "Owl: next generation system monitoring". In: *Proceedings of the 2nd conference on Computing frontiers*. CF '05. Ischia, Italy: ACM, 2005, pp. 116–124. ISBN: 1-59593-019-1. DOI: http://doi.acm.org/10.1145/1062261.1062284. URL: http://doi.acm.org/10.1145/1062261.1062284 (cit. on pp. 40, 41).

[65] A. Schupbach et al. "Embracing diversity in the Barrelfish manycore operating system". In: *Proceedings of the Workshop on Managed Many-Core Systems* (2008) (cit. on p. 27).

[66] Filippo Sironi. "Design and implementation of an hot-swap mechanism for adaptive systems". MA thesis. Politecnico di Milano, 2010 (cit. on p. 6).

[67] Craig A. N. Soules et al. "System Support for Online Reconfiguration". In: *Proc. of the Usenix Technical Conference*. 2003 (cit. on p. 33).

[68] Brinkley Sprunt. "Managing The Complexity Of Performance Monitoring Hardware: The Brink Andabyss Approach". In: *International Journal of High Performance Computing Applications* 20.4 (2004), pp. 533–540 (cit. on p. 40).

[69] Brinkley Sprunt. "The Basics of Performance-Monitoring Hardware". In: *IEEE Micro* 22.4 (2002), pp. 64–71 (cit. on pp. 12, 40).

[70] P. Stelling et al. "A fault detection service for wide area distributed computations". In: *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on.* 1998, pp. 268 –278. DOI: 10.1109/HPDC.1998.709981 (cit. on p. 41).

[71] R. Sterritt and D.F. Bantz. "Personal autonomic computing reflex reactions and self-healing". In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 36.3 (2006), pp. 304 –314. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2006.871592 (cit. on p. 64).

[72] Andrew S. Tanenbaum. *Modern Operating Systems.* 2nd. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130313580 (cit. on pp. 13–20).

[73] Linus Torvalds. *The Linux Kernel Archives.* [Online] Available: http://kernel.org/. 2011 (cit. on pp. 47, 56, 79).

[74] Princeton University. *Parsec benchmark suite website.* [Online] Available: http://parsec.cs.princeton.edu/. 2011 (cit. on p. 105).

[75] Various Authors. *The MIT Angstrom Project: Universal Technologies for Exascale Computing (project home page).* [Online] Available: http://projects.csail.mit.edu/angstrom/index.html. 2011 (cit. on pp. 32, 36).

[76] David Wentzlaff and Anant Agarwal. "Factored operating systems (fos): the case for a scalable operating system for multicores". In: *SIGOPS OSR* 43 (2 2009), pp. 76–85. ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/1531793.1531805. URL: http://doi.acm.org/10.1145/1531793.1531805 (cit. on pp. 2, 23–25, 36).

[77] David Wentzlaff et al. "An operating system for multicore and clouds: mechanisms and implementation". In: *Proceedings of the 1st ACM symposium on Cloud computing.* SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 3–14. ISBN: 978-1-4503-0036-0. DOI: http://doi.acm.org/10.1145/1807128.1807132. URL: http://doi.acm.org/10.1145/1807128.1807132 (cit. on p. 25).

[78] Paul E. West et al. "Core monitors: monitoring performance in multicore processors". In: *Proceedings of the 6th ACM conference on Computing frontiers.* CF '09. Ischia, Italy: ACM, 2009, pp. 31–40. ISBN: 978-1-60558-413-3. DOI: http://doi.acm.org/10.1145/1531743.1531751. URL: http://doi.acm.org/10.1145/1531743.1531751 (cit. on pp. 40, 41).

[79] Silas B. Wickizer et al. "Corey: An Operating System for Many Cores". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08).* San Diego, California, 2008. URL: http://www.usenix.org/events/osdi/tech/full_papers/boyd-wickizer/boyd_wickizer.pdf (cit. on pp. 9, 24, 26, 27).

[80] Robert W. Wisniewski and Bryan Rosenburg. “Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems”. In: *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. SC ’03. New York, NY, USA: ACM, 2003, pp. 3–. ISBN: 1-58113-695-1. DOI: `http://doi.acm.org/10.1145/1048935.1050154`. URL: `http://doi.acm.org/10.1145/1048935.1050154` (cit. on p. 41).

[81] Robert W. Wisniewski et al. “Performance and environment monitoring for whole-system characterization and optimization”. In: *In Proc. of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC), Yorktown Heights*. 2004, pp. 15–24 (cit. on p. 42).

[82] Feng Xian, Witawas Srisa-an, and Hong Jiang. “Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs”. In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOPSLA ’08. Nashville, TN, USA: ACM, 2008, pp. 163–180. ISBN: 978-1-60558-215-3. DOI: `http://doi.acm.org/10.1145/1449764.1449778`. URL: `http://doi.acm.org/10.1145/1449764.1449778` (cit. on p. 51).

[83] V. Yodaiken. “The RTLinux Manifesto”. In: *Proc. of The 5th Linux Expo, Raleigh, NC*. Mar. 1999. URL: `http://citeseer.ist.psu.edu/yodaiken99rtlinux.html` (cit. on pp. 21, 49).

November 28, 2011

Document typeset with $\LaTeX$