POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

Contention-Aware Thread Mapping driven by Self-Aware Synchronization Mechanisms

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Prof. Giovanni SQUILLERO

Correlatore: Ing. Filippo SIRONI

Tesi di Laurea di:

Matteo CARMINATI

Matricola n. 746551

Anno Accademico 2010–2011

To the ones who care about me.

Sincerely thankful,

MC

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AcOS Autonomic Operating System

API Application Programming Interface

CHANGE Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments

CIL Common Intermediate Language

CFS Completely Fair Scheduler

CPU Central Processing Unit

CSAIL Computer Science and Artificial Intelligence Laboratory

DEI Dipartimento Elettronica e Informazione

DI Distributed Intensity

DIO Distributed Intensity Online

DINO Distributed Intensity NUMA Online

DMS Distributed Shared Memory

DRAM Dynamic RAM

DTS Digital Thermal Sensors

DVFS Dynamic Voltage and Frequency Scaling

ECLiPSe ECRC Common Logic Programming System

EDF Earliest Deadline First

FAST Futex Aware Scheduling Technique

FCFS First-Come First-Served

FOL First Order Logic

fos Factored Operating System

FPGA Field Programmable Gate Array

Futex Fast User-space muTEX

GID Group IDentifier

GNU GNU's Not Unix

GPU Graphics Processing Unit

HDD Hard-Disk Drive

HPC Hardware Performance Counter

HR High-Resolution

HRM Heart Rate Monitor

IBM International Business Machines Corporation ™

HT Hyper-Treading

IHS Integrated Heat Spreader

IPC Inter Process Communication

ISA Instruction Set Architecture

I/O Input/Output

LLC Last Level Cache

MAPE-K Monitoring, Planning, Analyzing, Executing with shared Knowledge

MIPS Microprocessor without Interlocked Pipeline Stages

MIT Massachusetts Institute of Technology

NIC Network Interface Controller

NUMA Non-Uniform Memory Access

ODA Observe Decide Act

OS Operating System

OTT Object Translation Table

PAFS Performance-Aware Fair Scheduler

PDPA Performance-Driven Processor Allocation

PID Processor IDentifier

POSIX Portable Operating System Interfaces for uniX

PMU Performance Monitoring Unit

QoS Quality of Service

RAM Random-Access Memory

RCS Resource Constrained Scheduling

RDSL Rotating Staircase Deadline Scheduler

RMS Rate Monotomic Scheduling

RR Round Robin

SEEC SElf-awarE Computational model

Sefos Self-aware factored operating system

SJF Shortest Job First

SKB System Knowledge Base

SMART Scheduler for Multimedia And Real-Time applications

SMMP Shared-Memory symmetric Multi-Processor

SMP Symmetric Multi-Processor

SMT Simultaneous Multi-Threading

SPLASH Stanford Parallel Applications for Shared Memory

SPN Shortest Process Next

SRAM Static RAM

SRTN Shortest Remaining Time Next

SSD Solid-State Disk

TLB Translation Look-aside Buffer

Summary

Nowadays, the power and the complexity of computing systems are evolving and increasing at an unprecedented rate. The advantages of highly-parallel systems could benefit an enormous variety of fields. However, the growing complexity is making it unfeasible for the average programmer to weight all the constraints and optimize the system for a wide range of machines and scenarios. The burden on programmers is noticeable and many research efforts were spent in addressing this issue. Clearly, it is not feasible to rely on human intervention to tune a system: conditions change constantly, rapidly, and unpredictably. It would be desirable to have the system automatically *adapt* to the mutating environment.

A new paradigm is to be explored for these systems to be developed. Self-adaptive systems seem to be the answer to most of the problems previously described. They adapt their behavior and resources to automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. Therefore, this kind of systems needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions: implementing the Observe Decide and Act control loop.

The research work presented in this document aims at augmenting the GNU/Linux operating system with autonomic features. The idea is to make the system aware of the level of data contention among different tasks and to allow it to take smart decisions about how to actually map them on the

cores. Theoretical concepts about synchronization methods, memory hierarchy, and task scheduling and mapping help in sustaining that moving threads with high contention on the same core can lead to a reduction of the tasks execution time. To put this design into practice, the following contributions were made:

- a monitoring infrastructure able to quantify the lock contention among threads was implemented;
- an adaptation policy to smartly move tasks onto cores was designed;
- the modifications to the kernel of the Linux operating system, in order to include this policy, were made.

The remainder of this dissertation is organized as follows. Chapter 1 better introduces the work developed for this thesis, by clarifying the problem that has to be solved and defining a common terminology in order to provide a shared background. Chapter 2 describes the context in which this work was born, with a deep introduction to the autonomic computing field. The attention is then focused on the state of the art on topics directly related to this thesis: monitoring infrastructures and self-aware scheduling algorithms. The theoretical aspects behind the designed system and the developed framework are reported in Chapter 3, while the details of the actual implementation are specified in the following Chapter 4. In order to validate the proposed approach some experiments were performed: the results are reported and commented in Chapter 5. Last, Chapter 6 sums up the contributions of this work of thesis, proposing some interesting future development.

Sommario

La complessità e la potenza dei comuni dispositivi di calcolo cresce di giorno in giorno: processori dotati di più core si possono trovare facilmente nei computer portatili, architetture complesse ed eterogenee sono ormai diffuse nei tablet, nei telefoni cellulari e nelle game console di ultima generazione. Tuttavia, questa versatilità e questa potenza di calcolo devono essere controllate e gestite debitamente affinchè non vengano sprecate: è questa una delle sfide più impegnative nella progettazione di un sistema di questo tipo. Questa è anche una delle ragioni principali per cui il controllo delle risorse di un sistema non può essere interamente delegato all'utente umano: esso non è adatto a svolgere tale compito, a causa dei cambiamenti rapidi, inevitabili e spesso impredicibili delle condizioni in cui si trovano sia il sistema stesso sia l'ambiente in cui esso "vive". I dispositivi devono essere in grado, raccogliendo informazioni su se stessi e sull'ambiente, di adattarsi, in ogni istante, per raggiungere obbietivi prestazionali e di sicurezza.

Affinchè tale idee possano essere integrate in un sistema reale, occorre esplorare nuovi paradigmi di progettazione e design. Un sistema in grado di fare quanto descritto è noto in letteratura con il nome di sistema adattativo (autonomico o *self-aware)*. I sistemi adattativi rappresentano la risposta a molti dei problemi posti nel paragrafo precedente: essi sono in grado di adattare il proprio comportamento e le risorse di cui dispongono in modo automatico, e con l'obbiettivo di raggiungere, nel miglior modo possibile,

i propri scopi, nonostante un ambiente in continua evoluzione. Per far ciò un sistema deve essere in grado di monitorare se stesso e l'ambiente circostante, di riconoscere gli eventi significativi, di determinare quali siano le migliori azioni da intraprendere e di metterle in pratica. Questo paradigma distingue chiaramente tre fasi: una di monitoring, una decisionale e una in cui le azioni sono realizzate. Tale paradigma prende il nome di ciclo di controllo ODA (Osserva – Decidi – Agisci).

Questa categoria di sistemi è oggetto di questo elaborato di tesi e più in generale, del progetto CHANGE in cui questo lavoro si sviluppa. L'idea su cui si basa CHANGE, in particolare AcOS (il sistema operativo adattativo che si propone di realizzare) è l'integrazione del ciclo ODA all'interno di un sistema operativo. Affinchè il sistema possa osservare il proprio stato sono necessari dei *monitor*: per le prestazioni delle applicazioni, per il consumo di potenza del sistema o, ancora, per la temperatura dei core o per la contesa delle risorse. Con queste informazioni a disposizione, un *motore decisionale* (basato su euristiche, piuttosto che su tecniche di machine learning) deve in grado di prendere delle decisioni riguardo a quali siano le azioni più adatte da intraprendere per raggiungere gli obbiettivi sia delle diverse applicazioni che del sistema stesso. Agli *attuatori* è affidato il compito di tradurre in pratica le decisioni prese durante la fase precendente.

Nello specifico, il lavoro di ricerca presentato in questo documento ha lo scopo di inserire caratteristiche autonomiche all'interno del sistema operativo GNU/Linux. Intuitivamente, l'idea è quella di fare in modo che il sistema operativo abbia ha disposizione informazioni che gli consentano di quantificare e valutare la contesa di dati condivisi tra diversi *thread* che eseguono il loro codice. Il sistema dovrà quindi sfruttare queste informazioni per migliorare le prestazioni di tali processi (prestazioni misurate in termini di tempo di esecuzione), grazie a degli spostamenti intelligenti di *task* sui processori disponibili. Analizzando dal punto di vista teorico il funziona-

mento dei metodi di sincronizzazione utilizzati, della struttura della gerarchia di memoria implementata nei moderni processori, e degli algoritmi di scheduling e mapping del sistema operativo, è possibile sostenere che posizionare thread che si scambiano molto spesso informazioni e contendono per gli stessi dati su uno stesso processore, può portare a delle riduzioni nei tempi di esecuzione. I contributi al sistema operativo in oggetto, necessari affinchè quanto descritto possa essere effettivamente implementato, sono i seguenti:

- un'infrastruttura che permetta di monitorare e quantificare il livello di contesa sui dati condivisi da diversi thread;
- una (o più) politica decisionale che, presi i dati forniti dal monitor, sia in grado di muovere intelligentemente i thread sui processori disponibili, con l'obbiettivo di minimizzare il loro tempo di esecuzione;
- la modifica del kernel del sistema operativo in oggetto, implementando in esso la politica decisionale progettata.

Il resto di questa tesi è organizzato nel modo seguente. Il Capitolo 1 propone un'introduzione al lavoro svolto. In particolare il problema da affrontare viene esposto nei dettagli, sottolineandone i punti più critici. Nel seguito sono introdotte le definizioni necessarie a costruire un vocabolario condiviso riguardo gli argomenti alla base di questa tesi, con particolare riferimento agli algoritmi di scheduling e mapping dei processi, alla struttura gerarchica della memoria implementata nelle moderne architetture e ai meccanismi di sincronizzazione utilizzati nella scrittura di applicazioni multi-thread. In questo capitolo è data anche una breve introduzione al campo dei sistemi operativi autonomici.
Tuttavia, questo concetto è meglio approfondito nel Capitolo 2 dedicato alla definizione del contesto in cui questa tesi si colloca. Come detto, il concetto di sistema operativo autonomico viene definito da un punto di vista

puramente teorico; i più importanti esempi di sistemi operativi autonomici sono quindi analizzati. Il resto del capitolo presenta lo stato dell'arte per gli argomenti principali di questo lavoro: le infrastrutture di monitoring e gli algoritmi di scheduling adattativi. In entrambi i casi, l'accento è posto su due quantità in particolare: performance, da cui questo lavoro trae ispirazione, e la contesa dei dati, che questo lavoro tratta direttamente.

Il Capitolo 3 scende nello specifico degli aspetti teorici che supportano il lavoro svolto. Vengono presentate le modifiche che andranno introdotte affinchè l'attuale sistema di monitoring possa essere esteso per offrire informazioni sulla contesa dei dati condivisi. Sono inoltre descritte le politiche decisionali che permettono di muovere intelligentemente i thread , ottenendo delle riduzioni nei loro tempi di esecuzione.

I dettagli implementativi del lavoro sono presentati nel Capitolo 4. Sono descritte le modifiche al sistema di monitoring, apportate sia alla sua implementazione all'interno del kernel, che alla libreria che ne espone le funzionalità in user-space. Viene presentata la semplice libreria di locking debitamente instrumentata in modo da permettere l'implementazione dell'approccio proposto. Vengono infine proposti i dettagli dell'implementazione della politica incaricata di muovere i task sui processori, evidenziando in particolare le criticità incontrate.

Il Capitolo 5 mostra i risultati ottenuti con il sistema operativo debitamente modificato. In particolare, viene analizzato l'overhead dell'instrumentazione, e presentati alcuni esempi che permettono di comprendere il funzionamento del sistema e di valutarne le capacità.

Infine, il Capitolo 6 termina questo elaborato, riassumendo i punti cruciali del lavoro svolto e proponendo alcuni interessanti sviluppi futuri.

Chapter 1

Introduction

The power and the complexity of computing systems are evolving and increasing at an unprecedented rate: multi/many-core processors can be easily found in servers, desktops and laptops systems, complex and heterogeneous architectures are nowadays widely diffused in tablets, mobile phones, and game consoles [1]. On one hand, the advantages of highly-parallel systems could benefit an enormous variety of fields. On the other hand, the growing complexity is making it unfeasible for the average programmer to weight all the constraints and optimize the system for a wide range of machines and scenarios [2]. Even though technologies have improved, making a system perform at its best is a non-trivial task. The burden on programmers is noticeable and many research efforts were spent in addressing this issue. Clearly, it is not feasible to rely on human intervention to tune a system: conditions change constantly, rapidly, and unpredictably. It would be desirable to have the system automatically *adapt* to the mutating environment [3].

A common believe is the need for new paradigms to be explored and for new frameworks to be developed. Among those, self-adaptive systems seem to be the answer to most of the problems previously described [3]. Self-Aware Adaptive computing systems adapt behavior and resources to

automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. Therefore, this kind of system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions.

The work described in this document is strictly related to the concepts introduced above, and this chapter serves as an introduction to the whole document. In particular, Section 1.1 gives an high-level description of the problem this thesis aims at overcoming, while the basic concepts needed to understand the issues and the implications described in the following chapters are defined in Section 1.2.

1.1 Problem Statement

The design of a self-aware system able to abstract from low-level architecture details and capable of dynamically adapt to them, taking away this burden from the user, is a complex engineering problem. A wide range of issues is to be taken into consideration while thinking at its structure: among them the problem of resources allocation. Run-time information can be exploited in order to better perform an hard problem such as resource allocation. The term *resource* is very general: it can refer to the number of cores, to the working frequency of the cores, to the quantity of memory, and so on. In the same way, different types of *quantities* can be considered and monitored in order to make the system aware of itself and able to better perform: cores temperature, power consumption, applications performance, ... By coupling one (or more) resource(s) with one (or more) quantity(ties) many different aspects of self-adaptability can be implemented.

The research work presented in this thesis aims at exploiting information about resource contention (focusing on the contention of shared data among tasks) in order to distribute more efficiently the applications on the

cores available in the system. The task of moving the executing entities (technically called *tasks*) on the available processors is commonly known with the name of *task mapping*, and is usually coupled with the *task scheduling* problem: a brief overview of this concepts and some basic definitions are given in Section 1.2.2. The scheduling algorithms implemented in modern operating systems running on multi-core architectures exploit, as primary strategy for placing tasks on cores, *load balancing* [4]. This means that the scheduler tries to balance the runnable tasks across the available resources to ensure fair distribution of CPU time. However, this approach completely neglects the fact that a core is not an independent processor, but rather a part of a larger on-chip system, sharing resources with other cores. It has been documented in literature (see [5, 6] for further details) that the performance of a task can vary greatly depending on which threads run on the other cores of the same chip: this is especially true if several cores share some memory components. For this reason, an introduction to the hierarchical structure of memory in modern architectures is needed and is given in Section 1.2.3.

Nowadays, processes schedulers do not take the non-uniform sharing overheads into account. Threads that heavily share data will not typically be co-located on the same chip, resulting in many high-latency inter-chip communications. If the Operating System (OS) can detect the thread sharing pattern and schedule the tasks accordingly, then tasks that intensively exchange information could be scheduled on the same core and, as a result, the communication overhead would be reduced by exploiting the processor cache levels. In order to provide the necessary information about resource contention to the OS, a monitoring infrastructure must be designed. Programming languages and libraries support mechanisms to spawn threads and to make communication and synchronization between them possible (threads synchronization methods are presented in Sections 1.2.4). The ap-

proach embraced in this work consists in instrumenting a user-space locking library in order to provide the OS enough information to understand how much threads share information, i.e. how much they contend for a data. This information is then used within the Linux kernel to affect the work of the process scheduler, by influencing the mapping of the tasks on the cores, in order to reduce data contention, thus improving applications performance in term of their execution time.

The long term idea is to integrate the developed framework in a completely self-aware operating system, named Autonomic Operating System (AcOS), and born within the Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) research group at Politecnico di Milano. This framework should represent only a single *service* within the whole operating system, i.e. it should provide only one of many possible autonomic capabilities to the OS. This scenario introduces the problem of efficiently orchestrating the available services, in order to meet the, possibly conflicting, goals of the different applications or of the system itself. This topic was addressed in previous research works [7] and falls beyond the scope of this document.

1.2 Background Definition

In order to better understand the motivations behind the investigation of the problem described in the previous section and the solution that was found, a clearer explanation of the context in which this work has been developed is needed. First of all, a definition of autonomic computing is to be given and its fundamental pillars must be introduced (Section 1.2.1). The attention is then focused on topics directly related to the ones this research work is based on: the problem of task scheduling and mapping (Section 1.2.2) and the hierarchical memory organization of modern computer

architectures (Section 1.2.3). Last, data sharing and synchronization methods are investigated in Section 1.2.4.

1.2.1 Autonomic Computing

The research work presented in this document finds its natural location in the field of *autonomic* or *self-adaptive computing*. This term was firstly coined by Paul Horn in 2001 [2], deliberately referring to biological self-adaptation mechanisms. In particular, the analogy is made taking into account the nervous system of living beings, able to control common body actions and parameters, taking away this burden from the conscious part of the brain. Inspired by the same idea, autonomic systems should be able to adjust and manage their *vital* parameters, taking actions autonomously (monitoring the internal state of the system and the surrounding environment) and not asking the user to take care about these problems.

While formalizing the definition of an autonomic system in his *manifesto* [2], Horn listed some properties a system must own in order to show a *self-adaptive* behavior and called them *self-* properties*. How these self-* properties can be inserted into an autonomic system is not a straightforward topic: in literature, it is not possible to find a commonly shared model able to solve this issue, but there is no doubt a new system design paradigm must be introduced. This new paradigm is called *autonomic control loop* [2] and various definitions and description of this control loop can be found. All these description share a common basic idea: the system must somehow be able to monitor itself and the relevant element of the surrounding environment, reason on these data in order to take a decision according to its goals, and put into practice the computed decision by properly tuning its parameters.

For the purpose of this thesis, it is interesting to investigate how the autonomic loop can be embedded in a specific system such as a computing

system. First of all, monitors and actuators are to be identified, then a policy is to be designed to properly use the available information and effectively exploit the actuators.

In a common computing system, there are many quantities their monitoring could prove to be interesting: from the temperature of the cores to their power consumption, from the throughput of the system to its latency. Looking at the problem description provided in Section 1.1, it is clear that the quantity that is to be monitored to solve it must provide hints about how much variables are contended between different tasks.

1.2.2 Task Scheduling and Task Mapping

A topic which is crucial for this research and needs to be investigated is how tasks are scheduled for execution by the operating systems and how they are located on the available resources. Having a clear idea of these subjects is fundamental to realize how the behavior of the overall computing system can be modified by acting on them.

Preliminary Definitions

When dealing with operating systems, it is worth having a clear idea of some elementary concepts. In particular [8, 9]:

- **Process.** A *process* is an abstraction of a running program which represents an executing program, including its code, its data, and all the information about its execution status.
- **Thread.** A thread of execution, usually shortened to just *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 same address space, open files, and, in general, the resources assigned to the process.

- **Task.** A *task* is a schedulable entity (either a process or a thread).

While the first two definitions are universally accepted and adopted (since proposed in the Portable Operating System Interfaces for uniX (POSIX) standard), the last one is specifically related to the Linux kernel. However, the definition of what a *task* is, is fundamental to understand the following chapters and it is consistent with the context in which this thesis is developed. Summing up, from the point of view of the process scheduler and of the thread mapper, processes and threads are perfectly equal and they can be both called tasks.

Now, definitions of both process scheduling and thread mapping are needed [10].

- **Process Scheduling.** The problem of process scheduling can be traced back to a more general problem known with the name of *Resource Constrained Scheduling (RCS)*. In a few words, it is a generic problem modelling a situation in which a set of activities must be completed by using a limited set of available resources in order to optimize one or more objective function(s).

- **Thread Mapping.** The thread mapping problem is quite orthogonal to the process scheduling one. As said, the latter problem consists in choosing which and how much to execute a task; the former one deals with moving the task chosen for execution on the available resources, taking into consideration their specific conditions. Examples of this conditions are units load, temperature, power consumption, ...

Process Scheduling

In the context of a computing system, the activities to be scheduled are what we called *tasks* and the available resources are the *execution units* or *cores*. In particular, the scheduling algorithm of an operating system is in

charge of choosing the execution order of the tasks and how much time to give them for execution.

The role of the scheduler became more and more important with the attempt for computing systems to give the illusion of multiple processes executing at the same time. 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 the execution of the running tasks on the available processor(s) [8, 9]. The decision power of the scheduler depends on the kind of multitasking adopted by the operating system. In *cooperative multitasking* the scheduler is not allowed to stop the execution of a task: it must wait the task itself to explicitly yield the assigned resource (the processor on which it is executing) [9]. Such a paradigm was adopted by oldest operating systems (Microsoft Windows up to 3.1 and Mac OS up to 9): it allows a simpler implementation, but relies on tasks good faith to avoid system *starvation*. To avoid this major drawback, *preemptive multitasking* was introduced in modern operating system (Linux and the most recent versions of Microsoft Windows and Mac OS) [9]. In this case, more power is given to the scheduler: it is allowed to suspend a task execution in favor of another task. A maximum execution time, called *quantum*, is given to each task before being preempted. While preemptive multitasking requires a more complex implementation, it allows to avoid malicious tasks to take control of the system or normal program to get stuck in infinite loops due to bugs, preventing the processing unit to be available for other tasks.

An interesting classification of the process schedulers can be done by looking at the goals they pursue. These goals are formalized by objective functions, depending on the specific scheduling environment [9]:

- *Batch* systems usually process a series of programs, or *jobs*, sequentially and without the need for manual intervention. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time

periods for each process are often acceptable, since they reduce process switches, improving performance.

- *Interactive* systems, as the name suggests, foreshadow a significant interaction between the system and the users. In this case preemption is essential to prevent a task to own the CPU for so much time to deny the service to the other users.

- *Real-time* systems run jobs which have to meet time constraints. Preemption in real-time systems is sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. Real-time systems can be further classified in:

  - *Soft* real-time: when time constraints are not that strict and the scheduler provides its best effort to meet the deadlines, but it does not ensure any sort of warranty.
  - *Hard* real-time: when time constraints are strict. An error must be returned by the scheduler if a deadline is missed.

Specific scheduling goals for each type of system can be formulated according to the scheduling environments classification given before. In the same way, among all the designed scheduling algorithms some are more suitable for a type of system, some others are more suitable for other type of systems. Table 1.1 summarizes these two important aspects of tasks scheduling. It does not claim to exhaust the topics, but only to give some hints about them. A comprehensive dissertation about scheduling is out of the scope of this document; scheduling algorithms that are strongly related to this thesis will be analyzed in the next chapter, while you can refer to [8, 9] for a first introduction.

	Goals	Algorithm
All Systems	Fairness Policy Enforcement Balance	
Batch Systems	Throughput Turnaround time CPU utilization	First-Come First-Served Shortest Job First Shortest Remaining Time Next
Interactive Systems	Response time Proportionality	Round Robin Priority Scheduling Multiple Queues Shortest Process Next
Real-Time Systems	Meeting deadlines Predictability	Rate Monotomic Scheduling Earliest Deadline First

Table 1.1: Scheduling goals and algorithms according to the scheduling environment.

Thread Mapping

The problem of thread or task mapping arose later in the operating system design field, due to the relative youth of multi-core and multi-chip architectures. Besides the choice of which task has to run and when, it is important to decide where it has to execute. The performance of current shared-memory multi-processors systems heavily depends on the allocation of cores to parallel applications, especially in Non-Uniform Memory Access (NUMA) systems [11]. Performing the core allocation without taking into account some specific characteristics of the executing tasks (e.g. maximum speed-up or average parallelism) or the actual conditions of the processors, can result in a bad system exploitation. Two simple examples: allocating a high number of processors to a parallel application with small speed-up will result in a loss of efficiency; running a great number of tasks

on a single core, while leaving the other available ones unloaded will result, again, in a loss of processor utilization.

The latter example introduces a problem which is solved, in modern operating systems, by introducing a new component in the process scheduler, called *load balancer*. The goal of a load balancer is for each processor to perform an equitable share of the total work load. In literature [12], it is possible to find a clear distinction between:

- *static* load balancing: relies on an off-line a priori estimation of work distribution, so that a programmer can build load balancing right into a specific applications program.

- *dynamic* load balancing: refers to the case in which no a priori estimation of load distribution is possible and load information is available only during actual program execution.

Unfortunately the simplest form of load balancing, in which tasks requiring differing times for completion are to be as equally distributed as possible between n processors, is clearly equivalent to the partition problem, thus it is a NP-Complete problem as well [9]. As a result of this fact, research on dynamic load balancing (the one implemented in modern OSes) has focused on suboptimal procedures that use local information in a distributed memory architecture. Generally speaking, these procedures describe rules for migrating tasks from overutilized processors to underutilized processors. Tradeoffs exist between achieving the complete balance of the load and the communications costs associated with migrating tasks.

The Linux kernel implementation of the load balancing was deeply investigated and considered as an example for a possible implementation for the development of this work. For this reason, it is further analyzed in Chapter 2.

1.2.3 Memory Hierarchy

Memory is a fundamental resource in a computing system, used, for example, for data/code storing and for Inter Process Communication (IPC): the OS is in charge of managing this important resource. The OS should be able to provide the programmer with an infinitely large, infinitely fast memory abstraction, that is also *non-volatile* (i.e. it does not lose its contents when the electronic power fails). Unfortunately, even if the pace of technological development is getting faster and faster, such a memory is not available. Consequentially, most computing system relies on a *memory hierarchy*, exploiting different types of memories (with different speed/storage capability ratio), to provide the requested abstraction [9].

Figure 1.1 shows an intuitive representation of the different memory levels in a multi-core, multi-chip computing architecture. Only few registers are directly available on the same chip the core is on. Thus, few levels of small, very fast, expensive, volatile *cache* memory, some gigabytes of medium-speed, medium-price, volatile main memory (sometimes referred as Random-Access Memory (RAM), inaccurately), and terabytes of slow, cheap, non-volatile disk storage memory are necessary. Network storage represents an highest level in the hierarchy level; however, for the purpose of this work, the attention is limited to the previously presented levels.

- **Processor Registers.** Processor registers are at the top of the memory hierarchy, providing the fastest way to access data (usually only 1 CPU cycle) [13]. They offer a small amount of storage capability (the most common registers are 8-bit, 32-bit or 64-bit capable) and come in a limited number (from a few tens in x86, x86-64 and ARM architectures to more than two hundred on the high-performance Intel Itanium architecture) [14].

- **Cache Memories.** The slow access speed to the main memory repre-

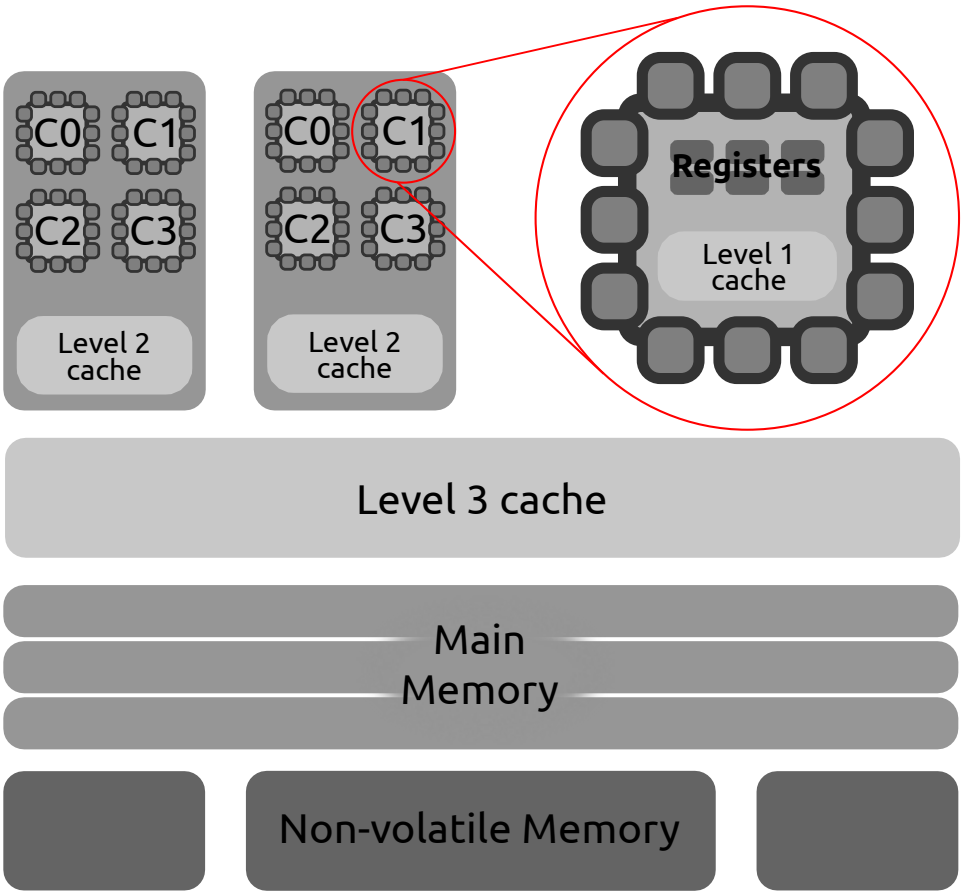

Figure 1.1: Memory hierarchy for a multi-core, multi-chip architecture.

sents a bottleneck in modern computing system [13]. A solution to this problem is the introduction of one or more levels of cache memory. Cache memories are small and volatile memories, but allow a high access speed (usually less than one hundred clock cycles).

- **Main Memory.** In contrast with caches, the main memory may take hundreds of cycles to retrieve the requested data, but is way more capable (up to tens of gigabytes). Caches and main memory are both RAM (classified into SRAM and DRAM [13]), thus volatile memories: their state is lost or reset when power is removed from the system.

- **Non-volatile Memory.** The lowest level of the memory hierarchy is

occupied by persistent storage devices, which maintain the information stored in them even when they are not connected to power. Many technologies are used to produce storage devices: from older but more reliable Hard-Disk Drive (HDD) to newer and faster Solid-State Disk (SSD). Non-volatile storage devices are extremely slow in performing read and write operations if compared with the other levels of the hierarchy: this is the reason why all this hierarchy was created.

For the purpose of this thesis, it is worth better understanding how caches actually work and how they can be exploited in order to improve applications performance.

Cache Memories

The effectiveness of the introduction of different levels, i.e., a hierarchy, of cache memories relies on computer programs peculiarity to obey to the *principle of locality*:

- *temporal* locality: if a particular memory location is referenced by a processor during the i-th cycle of execution, then with high probability the same memory location will be accessed during the execution cycle $(i + p)$ (where p is a small enough positive integer);
- *spatial* locality: if the data located at the address i is accessed by the processor, then with high probability the data located at the address $(i + q)$ will be accessed too (with q being a small enough non-zero integer).

In order to exploit this locality principle, caches employ buffering to reuse commonly occurring items [15]. When the Central Processing Unit (CPU) finds a requested data item in the cache, a *cache hit* takes place. A *cache miss* occurs when the CPU does not find a data item it needs in the cache. In this case, a fixed-size collection of data containing the requested word,

called *block* or *cache line*, is retrieved from the main memory and stored in the cache. This approach allows the cache to exploit both temporal locality (since the processor is likely to need this word again in the near future) and spatial locality (there is high probability that the other data in the block will be needed soon). Obviously, there is a performance improvements if the majority of the CPU data requests results in a cache hits, and the communication overhead with the main memory is reduced as much as possible.
A new issue now arises: larger caches have better hit rates but longer latency. Thus, the trade-off between cache latency and hit rate is to be considered and analyzed. To address this trade-off, multiple levels of cache were introduced, with small fast caches backed up by larger and slower caches. Multi-level caches generally operate by checking the smallest Level 1 (L1) cache first. If the result is a cache hit, then the processor can proceed at high speed. If the smaller cache misses, the next larger Level 2 (L2) cache is checked, and so on, until the main memory is reached. Modern architectures usually provides as many as three levels of on-chip caches [14].

L1 caches are usually dedicate to a single core, while L2 and L3 caches can be shared among different cores and chips. Multi-level caches can be *strictly inclusive*, in the sense that all data in the $L(n)$ cache must also be in the $L(n+1)$ cache, or *exclusive*, meaning that the same data cannot be both in $L(n)$ and $L(n+1)$ cache. Many other intermediate policies are implemented in commercial processors, but they have not a universally accepted name [15].

The cache and the main memory have the same relationship as the main memory and disk storage. In fact, not all the objects referenced by a program need to reside in main memory: some of them may reside on the disk. When a CPU references an item within a *page* (blocks the address space is broken into) that is not present in the cache or main memory, a *page fault* occurs. In this case, the entire page is moved from the disk to main memory,

with a consequently high overhead in terms of time (so high that usually the CPU switches to some other executable task while the disk access occurs).

1.2.4 Race Conditions and Synchronization Methods

The importance of synchronization methods become clear when thinking about shared memory applications, i.e., applications that share the entire address space or portion of memory with other applications during their execution. Shared resources require protection from concurrent access: if multiple threads/processes access and manipulate the same resource at the same memory location at the same time, the threads/processes may overwrite each other's changes or access data while it is in an inconsistent state [8]. These situations, where two or more thread/processes are reading or writing some shared data and the final result depends on who runs precisely when, are called *race conditions* [9]. Race conditions are source of instability and are usually hard to debug, due to the non-determinism they introduce. For this reason, race conditions are to be avoided when writing code.

When dealing with concurrency problems, it is important to identify in the source code the instructions that may create race conditions or deadlocks. That part of the program where the shared memory is accessed is called *critical region* or *critical section* [9]. To prevent problems when executing code inside a critical region, it is important for the programmer to ensure the code executes *atomically*: instructions complete without interruption as if the entire critical region was one indivisible instruction. Each OS offers different methods to provide atomicity: in the following paragraphs the attention is focused on both hardware and software synchronization methods. In the case of software solutions, POSIX compliant synchronization methods are presented [16].

Atomic Operations

The lowest level of synchronization primitives is represented by *atomic operations*. Atomic operations provide instructions that execute atomically: the OS assures that these instructions are completed without interruption. This is possible by disabling interrupts while their execution.

Two types of atomic operations exists: one operates on integer values, the other operates on individual bits. Most architectures contain instructions that provide atomic versions of simple arithmetic operations. Other architectures, lacking direct atomic operations, provide an operation to lock the memory bus for a single operation, thus guaranteeing atomicity [8]. Referring in particular to the two sets of interfaces for atomic operations supported by Linux, we have:

- *atomic integer operations*: these methods operate on a special data type, named `atomic_t`. The use of this special type ensures that the data types are not passed to any nonatomic functions and that clever but erroneous compiler optimizations are performed. Common uses of atomic integer operations are: counters implementation (through `atomic_inc()` and `atomic_dec()` functions) or atomically performing an operation and testing the result (e.g., `atomic_{sub|dec|inc}_and_test()`).

- *atomic bitwise operations*: unlike integer ones, these operations are architecture-specific and operate on generic memory addresses. Due to this reason, their arguments are a pointer (to whatever data type) and a bit number. Examples of atomic bitwise operations are `{set|clear|change|test}_bit()` or `test_and_{set|clear|change}_bit()`.

The use of atomic operations is to be preferred, when possible, over more complicated locking mechanisms, since the former ones, on most architec-

tures, incur less overhead in terms of time and less memory waste.

Fences

Fences are an hardware synchronization mechanism which allows to instruct the compiler not to reorder instructions around a given point, called *barrier* or *fence*. Occasionally, it is important that memory writes are seen by other code and by the outside world in the specific order the programmer intends. This is often the case with hardware devices but is also common on multiprocessing machines [8].conclusioni: concetto spaziale, fontana

The use of fences is quite wide: a *read memory* barrier ensures that no loads are reordered across its call. This means that no loads prior to the call will be reordered to after the call, and no loads after the call will be moved before the call. A *write memory* barrier functions in the same manner of a read memory barrier, but with respect to stores instead of loads. Last a simple *memory* barrier provides both a read and a write barrier. Special types of barrier are *conditional* barriers: it proved to be useful to have a read memory barrier but only for loads on which subsequent loads depend.

Spin Locks

The simplest software mechanism which allows to avoid race conditions is to have a single variable, shared among all the concurrent processes and called *lock variable*. This variable is initialized to 0 and can assume only two values: 0 and 1. When a process attempts to enter its critical region has to check the value of this variable: if the value of the lock variable is 0 it means there is no possibility of race conditions and the process can continue executing. Otherwise, if the value is 1, the process has to stop, waiting for the lock variable to change its value: continuously testing a variable until some value appears is called *busy waiting*. The synchronization method that uses busy waiting is called *spin lock*.

Spin locks are the most trivial type of lock and are used only when there is a reasonable expectation that the wait will be short. If not, more complex and efficient locks are needed.

Semaphores

A different synchronization approach is offered by *semaphores*. While tasks consume computing resources in waiting for a spin lock to be released, semaphores manage to avoid resource wasting in this sense. When a task attempts to acquire a semaphore that is unavailable, the semaphore places the task onto a wait queue and puts it to sleep [8]. This mechanism provides better processor utilization than spin locks because there is no time spent busy looping. However, semaphores have much greater overhead than spin locks: the programmer must take into consideration the execution context in order to select the most suitable synchronization method among the two. In particular, semaphores are well suited to locks that are held for a long time.

Another interesting feature of semaphores is that they allow for an arbitrary number of simultaneous lock holders. This number can be set at declaration time and it is called *usage count* or simply *count*. If the maximum allowed value for count is set to 1, the semaphore is called *binary semaphore* or *mutex*.

Mutexes

Binary semaphores are so widely used that they deserved a specific implementation and, thus, deserve some more investigation. In literature they are better known as *mutexes*, since they enforce mutual exclusion. They behaves similar to semaphores with a count of one, but they have a simpler interface, more efficient performance, and additional constraints on their use. This means that only one task can hold the mutex at a time, whoever

locked a mutex must unlock it, and recursive locks and unlocks are not allowed.

On one hand, preferring mutexes to semaphores is a matter of usage count. On the other hand, the reasons for using spin locks instead of mutexes are the same presented in the previous paragraph, writing about semaphores.

Condition Variables

Another synchronization device related to mutexes are *condition variables* [17]. Condition variables provide an efficient way to execute some code only when a flag is set, and pausing when the flag is not set, having the flag shared between tasks. Correctly, this can be done by protecting the shared flag with a mutex, but this implementation is not efficient, since the task function will spend lots of CPU whenever the flag is not set, checking and rechecking the flag, each time locking and unlocking the mutex. Condition variables allow to put the thread to sleep when the flag is not set, until some circumstance changes that might cause the flag to become set.

Barriers

Barriers allow to synchronize different threads by creating a *checkpoint* at which the calling thread shall block until the required number of threads has reached the same barrier. If a thread has to wait for other threads, it is put to sleep. When the last thread reaches the barrier, a signal is delivered to all the sleeping threads, awaking them and allowing them to continue their execution.

Table 1.2 sums up the main features of the different typologies of synchronization methods, trying to compare them and to underline when us-

ing one mechanism is better than using another.

Requirement	**Advantages**	**Disadvantages**	**Features**
Atomic Operations	Low overhead	Low expressiveness	Single lock holder
Fences	Low overhead	Low expressiveness	No instruction reordering allowed
Semaphores	CPU free during execution	High overhead	Multiple lock holders
Spin Locks	Low overhead, high expressiveness	CPU busy during execution	Single lock holder, suitable for short lock hold time
Mutexes	CPU free during execution, high expressiveness	High overhead	Single lock holder, suitable for long lock hold time
Condition Variables	CPU free during execution	High overhead	Conditional Execution
Barriers	Low overhead, CPU free during execution	Low expressiveness	Checkpoints creation

Table 1.2: Synchronization methods comparison.

Using synchronization methods is a solution to guarantee the correct execution of multi-threaded applications but, writing correctly synchronized code is hard and a new class of bugs may arise, called *deadlocks* [17]. A deadlock occurs when one or more threads are stuck waiting for something that will never be available. Two simple examples of deadlock situations are the following [8]:

- *self-deadlock*. If a thread of execution attempts to acquire a lock it al-

ready holds, it has to wait for the lock to be released. But it will never release the lock, because it is busy waiting for the lock, as shown in Table 1.3.

Thread 1
`Acquire lock A`
`Acquire lock A, again`
`Wait for lock A to become available`
`... deadlock ...`

Table 1.3: Self-deadlock pseudo-code.

- *deadly embrace*. Consider n threads and n locks; if each thread holds a lock that another thread wants, all threads will be stuck waiting for their respective locks to become available. Each thread is waiting for the other, and none of them will ever release its original lock; therefore, none of the locks will ever be available. The following Table 1.4 explains the problem in the case $n = 2$.

Thread 1	**Thread 2**
`Acquire lock A`	`Acquire lock B`
`Try to acquire lock B`	`Try to acquire lock A`
`Wait for lock B`	`Wait for lock A`
`... deadlock ...`	`... deadlock ...`

Table 1.4: Deadly embrace pseudo-code.

1.3 Summary

This first chapter introduced the research work that this document describes. First, an high-level description of the context and of the problem to

be solved were given (Section 1.1). Then, the background needed to fully understand the context was defined in Section 1.2. A brief discussion on the field of self-aware computing systems was done in Section 1.2.1, since further room is dedicated to them in the next chapter. The rest of the chapter analyzed the basic concepts related to the issue of processes scheduling and mapping (Section 1.2.2), to the hierarchical structure of memory in modern computing architectures (Section 1.2.3), and to processes synchronization methods (Section 1.2.4) in order to create a common terminology and make the remainder of the document more easily comprehensible.

Next chapter moves the attention on a more specific and detailed definition of the context within this work is developed. First of all, a description of the field of autonomic computing is given both from a theoretical an from a practical point of view (thus giving the fundamental definitions and presenting the more significant examples of autonomic operating systems). Then, the focus is on the state of the art related to the topics treated and developed in this thesis: process monitoring and self-aware task mapping and scheduling.

Chapter 2

Context Definition

The research work developed and described in this document finds its natural location in the field of *autonomic computing*. In particular, the aim of this work is to modify the Linux kernel in order to add autonomic capabilities to the GNU/Linux Operating System (OS). Before discussing the approach exploited to insert these capabilities into the chosen OS and its actual implementation, it is useful to take stock of the research already done on autonomic OSs and of the state of the art on system monitoring and self-aware scheduling.

In the last few years the awareness of the design of the most popular and traditional operating systems being outdated, grew considerably. New computing systems, made up with heterogeneous components and way more complex, require the design of the OS to be rethought. Contemporary OSes were conceived to manage uniform and cache-coherent systems, not considering the increasing number of programmable units available in modern architectures [18] and their growing heterogeneity [1].

One step in the OS redesign process is, exactly, the introduction of autonomic capabilities: the aspect is addressed in Section 2.1. In Section 2.2 particular attention is deserved to the state of the art on system monitoring and tracing (Section 2.2.1), fundamental for the system to know something

about its status and the status of the surrounding environment. Last, in Section 2.2.2, the field of *-aware tasks scheduling and mapping is deeply analyzed, particularly focusing on the topic of *contention*-aware scheduling and mapping, tightly coupled to the work presented later.

2.1 Autonomic Operating Systems

Taking into consideration autonomic capabilities while designing new OSes is not a mere academic exercise, but helps applications developers in avoiding explicitly handle parallelism, and explicitly consider energy efficiency, reliability and predictability issues [19]. New OSes should provide a further abstraction layer between the hardware and the programmer, hiding the complexity of the underlying system by self-managing their resources. The following sections address exactly this topic, first from a theoretical point of view, analyzing the principles that lead the design of a new autonomic operating system, then from the practical one, introducing the projects available in literature.

2.1.1 Design Principles

The *self-* properties* proposed by Horn in [2], were further investigated by Kephart and Chess [20] and put in a taxonomy, presented in [21] by Salehie. The different levels at which *Self-* properties* are considered, are shown in Figure 2.1 and presented below:

- Properties such as *self-awareness* and *context-awareness* are located at the lowest level of the hierarchy proposed: the *primitive level*. These two capabilities allow the ones in the higher levels of the hierarchy to appear. They refer to the ability of the system of monitoring and being aware of its self states and behaviors, and of its context, i.e. the operational environment.

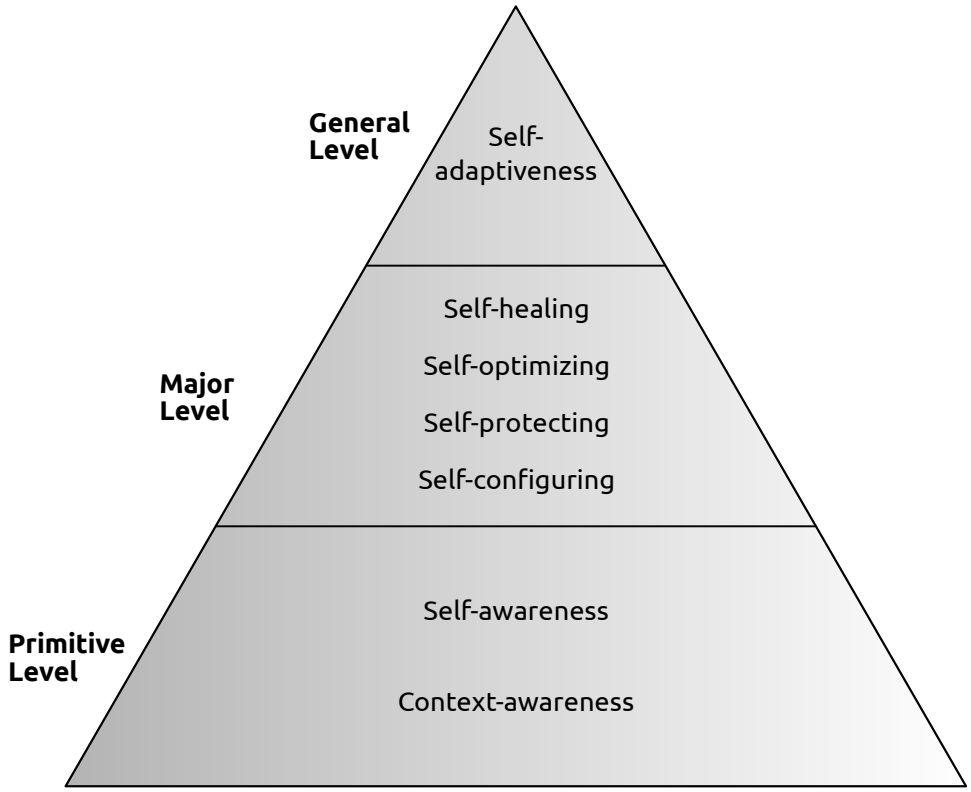

Figure 2.1: Self-* properties hierarchy.

- The main properties envisioned by Horn belong to a *major level* and, as already said, are the ones directly related to the human body self-adaptation capabilities to the changes in its status or in the environment it lives in. Due to their specific interest, these properties are listed and further analyzed in the following paragraphs.

  – *Self-configuring*. The capability of a system of installing, configuring and integrate different sub-modules automatically and dynamically. All this should be done in response to changes in the internal status of the system or in the external environment. High-level policies are to be specified in order to lead the system to the desired goal, without forcing constraints on how this goal is to be reached.

  - *Self-healing*. Directly linked to *self-diagnosis* and *self-repairing*, it is the ability of a system to react to localized problems both in software and hardware. It is achieved by firstly discovering and diagnosing faults, and then trying to fix them; proper actions can be taken also to prevent failures.
  - *Self-optimizing* is the capability of managing performance and resource allocation; more in general, to autonomously tune the parameters the system works on, in order to satisfy the different requirements. This property is also known with the name of *self-tuning* or *self-adjusting* [22].
  - *Self-protecting*. This capability allows the autonomic system both to detect and recover from the effects of malicious attacks, and to anticipate problems, taking actions in advance to avoid them or, at least, to mitigate their effects.

- The highest level, named *general level*, refers to properties which consider the whole system as a single entity. *Self-organization* and a plethora of self-* properties, which fall under the umbrella of *self-adaptiveness* (such as *self-management/government/maintenance/control/evaluation*), belong to this level.

So far, the basic definitions of autonomic computing were given and its fundamental pillars introduced: now, the steps a system must implement to show an autonomic behaviour deserve to be further investigated. In literature, it is not possible to find a commonly shared model able to solve this issue, but there is no doubt a new system design paradigm must be introduced. This new paradigm is called *autonomic control loop* [2]. Various definitions and description of this control loop can be found: the most common ones are described in the remainder of this section.

A first version of the autonomic control loop is described by Salehie again in [21]. A clear description of the stages the *Self-adaptation control loop* is made up of is given in Figure 2.2. The *Monitoring* stage allows the autonomic system to interface with self and with the environment, reading the data coming from the available sensors. These data are analyzed in the *Detecting* stage, asked for identifying *when* and *where* the system must change, according to current internal and external conditions. During the *Decision* stage, the system is in charge of deciding *what* is to be changed in the system and *how* these actions must take place. The mapping of actions into tasks, performed by actuators, is carried out in the *Acting* stage of the loop.

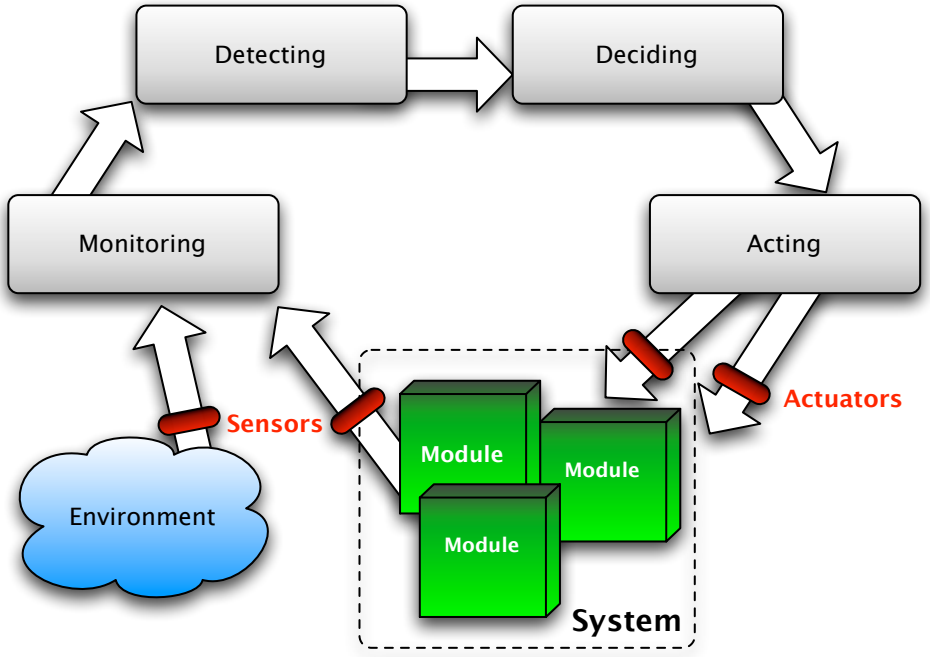

Figure 2.2: From [7] – Self-adaptation control loop.

This first interpretation of the autonomic control loop highlights the difference between the *when/where* and the *what/how* of a change, decoupling them in two different stages (*Detection* ad *Decision*, respectively). While sharing with this one the same basic ideas, other interpretations focus their

attention on slightly different details. The *Monitoring, Planning, Analyzing, Executing with shared Knowledge (MAPE-K)* loop [20] (Figure 2.3), for example, emphasizes the presence of a central entity storing the global knowledge about the system. *Monitoring*, *Analyzing*, *Planning* and *Executing* are the four stages the loop is constituted, with *K* standing for the shared knowledge, accessible from all the steps.

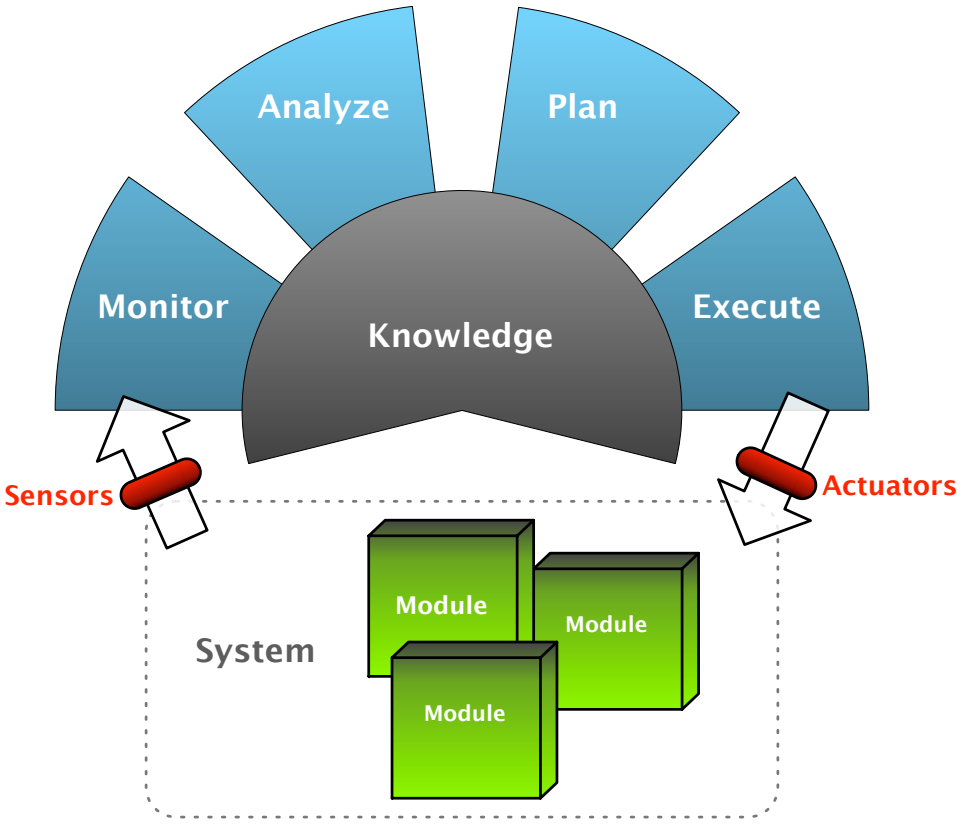

Figure 2.3: From [7] – *MAPE-K* control loop.

The *Observe Decide Act (ODA) loop* [23], shown in Figure 2.4, represents a third version of the autonomic control loop. Even if it can be considered equivalent to the other interpretations of the autonomic loop, the *ODA* loop better captures the essence of autonomic computing, by clearly dividing the system design in three simple and sharply distinct stages. Due to this reason, each single step of this loop deserves to be further investigated:

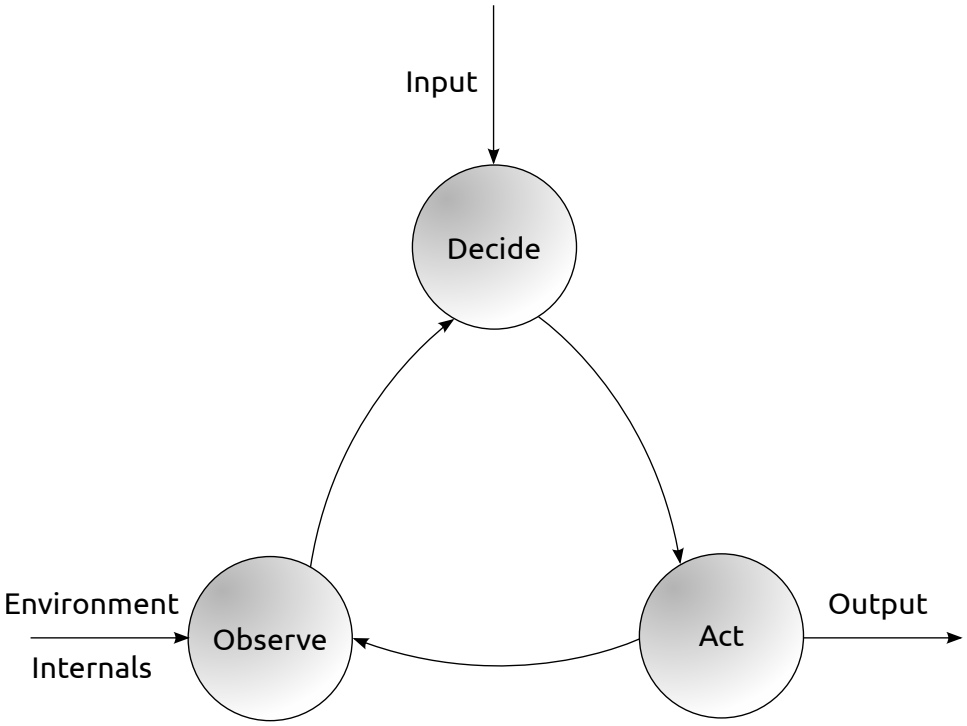

Figure 2.4: *ODA* control loop.

- **Observe**. The observation phase consists in sensing both the external environment and the internal behavior of all the sub-systems in order to maintain and update information about the *state* of the system. The sensing task is accomplished by *monitors*: thermometers, voltmeters and throughput meters are only a few examples of widely diffused monitors.

- **Decide**. This phase is performed taking into account the data obtained by the monitors and an high-level *goal*. The knowledge of the goal guides the logic of the system in coming up with a suitable *decision* which should approach the state of the system to the desired one.

- **Act**. Once the decision has been taken, it is put into practice in the acting phase through the actuators. Actuators are able to modify some

system parameters in order to alter its behavior.

Even if the presented autonomic loops come with different names and number of steps, they are all based on the same basilar concepts, as the reader can easily realize.

The following section lists some interesting research projects trying to insert autonomic capabilities in an operating system. Their analysis is interesting in order to understand their strong and weak points for the implementation of a new autonomic operating system.

2.1.2 Existing Projects

In literature, it is possible to find many examples of operating systems specifically designed to provide autonomic capabilities: some of them are more targeted towards multi- and many-core systems (fos – Section 2.1.2, Corey – Section 2.1.2, Sefos – Section 2.1.2), while others are more focused on supporting heterogeneous architectures (Barrelfish – Section 2.1.2, Helios – Section 2.1.2, K42 – Section 2.1.2).

Factored Operating System (fos)

The Factored Operating System (fos) [1] has been designed and implemented at the Massachusetts Institute of Technology (MIT) and, as anticipated, targets many-cores computing systems. In facts, the trend in computer architectures is heavily going in the direction of packing an increasing number of computer units on a single chip [24]. The belief this project is based on is that the real bottleneck in contemporary OSes is represented by the use of hardware locks and global cache-coherent shared memories. The novel design introduced by fos suggests to exploit the available parallelism by separating the execution resources of the operating system and the applications.

To obtain this separation, the OS was designed in three layer: a thin micro-kernel, at the basis; an OS layer, made up of servers providing typical system services; and an application layer, which makes use of the services offered by the operating system. By executing a portion of the micro-kernel on each core, the system is able to better control the access to hardware resources and to exploit caches for messages delivery, allowing servers and applications to communicate. Such an approach proved to be valid in system with an high number of cores (hundred and more) or in cloud-systems [25, 26].

Recent improvements of the operating systems blazed new trails toward the introduction of autonomic capabilities in fos [25]. First, 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. In this context, it is worth noting how, in the fos OS, the task scheduler has to deal with space (cores) multiplexing, instead of the classical time multiplexing problem. Second, the introduction of autonomic capabilities could improve the faults detection and recovery process: faults in system services must be detected by a watchdog process and handled by the name server by reassigning faulted communication channels.

Corey

As fos, Corey [27] is an experimental OS targeting towards multi-core systems and, as fos, it is developed at the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT. In contrast with fos, however, Corey implements a less radical approach, not requiring a computing system with hundreds of core to work properly. The specific goal of Corey is to improve applications scalability with reference to the number of cores, by better managing and exploiting processor caches and shared data struc-

tures. The idea underlying the implementation of such OS is the following: in order to avoid contention on data shared by different cores, new interfaces must be proposed to improve the management of these shared data, avoiding the use of shared structures unless strictly necessary. Three are the interfaces that have been implemented:

- *Address ranges*. They are an abstraction, provided by the kernel, corresponding to a range of virtual-to-physical memory mappings. Multiple ranges can be defined by an application; each of these ranges has to be mapped as *shared* or *private*. This flag allows to mark as shared only the data structures that really need sharing, reducing synchronization issues on data that are private to one thread. Comparing this interface with the classical paradigms implemented in common OSes, a new degree of freedom is introduced in the shared memory management.

- *Kernel cores*. This abstraction allows applications to declare that a specific core is to be dedicated to kernel functions. Hardware device managing or system calls execution are two simple examples of what dedicating a single core to a specific function means.

- *Shares*. This last interface offers to the application the possibility to create shared data and to specify at which level these data must be shared. As it was for address ranges, this interface introduces a further degree of freedom in the management of data contention and makes it possible to avoid unnecessary contention on data that need not to be shared.

Thanks to these new interfaces, the Corey OS proved to be more flexible if compared with fos, since it gives to applications the possibility of choosing the level of separation between the OS and application resources (while fos statically enforces this separation).

The main drawback of the described approach is the higher complexity of the exposed interfaces. Thus, the operating system has to rely on a wise use of such interfaces by the application: a future development of the system is the introduction of autonomic capabilities to reduce the exposed complexity [27].

The Angstrom Project

The Angstrom project [28] was born, again, in the CSAIL at MIT and aims at extending the already described fos OS, coupling it with a SElf-awarE Computational model (SEEC). SEEC introduces new autonomic features to the existing operating system, creating a new Self-aware factored operating system (Sefos), i.e., a self-aware operating system able to meet the challenges introduced by many-cores architectures.

As explained in the first section of this chapter, fos offers basic services (e.g., file system, memory management, network management), bounding them to specific cores. The aim of the Angstrom project is to create a new service in charge of introducing more autonomic features into fos. This new service is made available through SEEC, a framework which implements the typical ODA decision loop (see Section 1.2.1). The autonomic system, augmented with SEEC, executes and monitors itself using sensors. The system is able to react to the sensed conditions, taking decisions and acting to guarantee applications performance. The adopted monitoring interface is Application Heartbeats (further analyzed in Section 2.2.1). There is a decision engine acting on the system to set the values of each decision parameter. This decision engine exploits a control theory based control system, working on the following services to tune several parameters of the system:

- a *frequency scaler* which implements a Dynamic Voltage and Frequency Scaling (DVFS) policy to adjust the clock speed of the available cores;

- a *core allocator*, able to assign a subset of the system processors to the running processes;
- a *DRAM allocator* which efficiently manages multiple memory controllers (if more than one is available in the system) and assigns them to running tasks;
- a *power manager*, in charge of combining the previous services in order to directly affect the power consumption of the computing system.

However, when multiple decisions are to be taken at the same time, it is not clear how the coordination between the different actuation mechanisms available in the system takes place.

Analyzing the implementation of the system, some possible vulnerabilities/weak points can be identified. First of all, SEEC is completely written in user-space, limiting the actual effectiveness of the approach: the implementation of some services in kernel-space would be advisable and more effective. Notice also that Sefos relies on trusted actuators: no security checks are performed on the taken decision, thus malicious entities cannot be detected and deactivated.

Barrelfish

While the OSes analyzed up to now where focused on multi/many-core systems, an operating system devoted to improve scalability on heterogeneous architectures is Barrelfish [29]. When talking about heterogeneity, it can be classified at least in three different levels [30]:

- *Non-uniformity*: it refers to non-uniform memory architectures, for example Non-Uniform Memory Access (NUMA), and it is a characteristic more and more present in modern architectures, making the set of processing units similar to a network.

- *Core diversity*: is related to heterogeneity on multiprocessor architectures. Nowadays, common architectures contain processors that are exactly the same, thus homogeneous; Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) con be seen as *diverse cores* if used for specific tasks. However, it is normal to find heterogeneous cores in embedded systems, where often there are different specialized processing units.

- *System diversity*: is the most high level type of diversity and indicates diversity among the hardware components of different systems. A mobile phone and a internet server represent a significant example of system diversity: it is quite difficult for application programmers to write software required to run efficiently on diverse platform.

The main idea Barrelfish is based on is the design of a completely distributed operating system, made up of functional units using explicit message passing to communicate. This design technique allows the system to work properly and efficiently even if programmable units that cannot be made cache coherent with the rest of the system (such as GPUs and Network Interface Controllers (NICs)) or do not support shared memory at all are part of the architecture. This is possible trough a *multi-kernel* model [29] in which all the inter-process communications are managed explicitly.

The multi-kernel idea is exemplified in Figure 2.5. Here it is possible to see how separate entities execute on each programmable component of the system and asynchronously communicate by means of message passing. The main advantage of such an approach is the possibility to execute architecture-dependent code on each node, while leaving the operating system completely unaware of the specific architectures of the processors it is running on: with this meaning, the OS is considered to be *hardware-neutral*.

Barrelfish shows an autonomic behavior in the sense that exploits a System Knowledge Base (SKB) to make the operating system aware of its

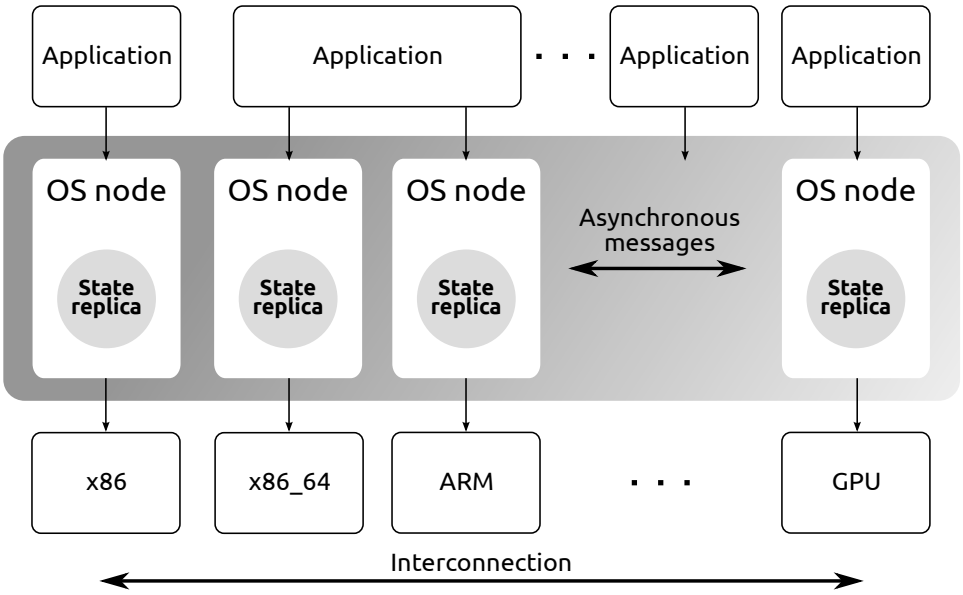

Figure 2.5: Multi-kernel model in the Barrelfish Operating System.

status. The knowledge base stores information about the hardware available in the system and its performance; information is maintained so that using a subset of First Order Logic (FOL) some reasoning is made possible [31]. This reasoning is exploited mainly by the process scheduler, as it is clear reading the design principles (described in [31]) which inspired its implementation:

- The System Knowledge Base can be exploited to take on-line decisions about the hardware on which processes are to be scheduled: this should permit to select the best policy on each node.
- While distributing resources, the scheduler must be aware of all applications workload and requirements. Applications can expose this information through a *scheduling manifest*, written in the ECRC Common Logic Programming System (ECLiPSe) language [32].
- In contrast with fos, the scheduling problem cannot be reduced to a mere spatial partitioning: time multiplexing is also needed on each

core.

- Different time scales can be identified in order to make the scheduling algorithm more effective: *long-term* placement of application on cores, *medium-term* resource allocation in reaction to application demands and *short-term* fine-grained per-core thread scheduling.
- Communication between the applications and the OS is needed to obtain efficient resource allocation: *dispatcher groups* are in charge of tuning system parameters to let applications meet their performance goals.

Along with the analysis of the OS design, a single drawback arose: as several of the previously described projects, Barrelfish relies too much in applications programmers' ability and good will.

Helios

Helios is an operating system designed in the Microsoft Research laboratories to simplify the task of writing, deploying, and tuning applications for heterogeneous platforms [18]. To achieve these goals, Helios introduces the concept of *satellite kernels*, which export a single, uniform set of OS abstractions across CPUs of disparate architectures and performance characteristics. This model is very similar to the one proposed by Barrelfish with the *multi-kernel* (Section 2.1.2). A satellite kernel can run indifferently on any programmable component having at least: a CPU, an (even little) amount of Random-Access Memory (RAM), a timer, an interrupt handler, and a mechanism able to catch exceptions. The listed constraints are not that loose, even if they appear to be so: GPUs, for example, are not allowed to run a satellite kernel since they are not usually equipped with timers or interrupt controllers. Authors rely on future generations of these hardware components for the obstacle to be overcome [18].

Through satellite kernels the heterogeneity of the underlying hardware is hidden to the application programmer, who can rely on the Application Programming Interface (API) and on the abstractions offered by the OS. Each satellite kernel can be considered a micro-kernel and is made up of a scheduler, and memory/namespace/communication managers. In particular, access to I/O services such as file systems are made transparent via remote message passing, which extends a standard micro-kernel message-passing abstraction to a satellite kernel infrastructure. The message-passing system is provided by the Singularity framework [33] and offers both *local* message passing (for communications within a single satellite kernel) and *remote* message passing (for communications between different satellite kernels). This framework allows to implement safe and efficient process software isolation and a fast zero-copy means of passing messages within the same address space (highly reducing the local message passing overhead).

Helios offers to the application programmer a unique system abstraction over heterogeneous hardware: this is possible by encapsulating the Instruction Set Architecture (ISA) of each processing unit and providing a unique programming language for the applications development. To do that a two-stage compilation strategy [34] is implemented: applications, written in `Sing#` (a derivative of the `C#` programming language), are first compiled into an intermediate language, called Common Intermediate Language (CIL) and part of the `.NET` framework, and then translated to the specific ISA of the node where are to be executed.
To simplify the process of application deploy and performance tuning, Helios exposes an *affinity* metrics to developers. Affinity provides a hint to the OS about whether a process would benefit from executing on the same platform as a service it depends upon. Processes are allowed to specify a [18]:

- *processes affinity*: indicates the coupling level between two processes.

It can be positive (two processes should run on the same satellite kernel – e.g., a driver and a process that uses it) or negative (preference for separate execution).

- *platform affinity*: indicates the preference for an application to run on a specific type of architecture (*Out-of-order x86* and *Vector CPU* are only two examples of architectures, a typical x86 processor and a GPU, respectively).

- *self-reference affinity*: indicates the ability of a process to efficiently scale-out its performance by running multiple instances of itself on different devices or NUMA domains.

K42

K42 [35] is an open source research operating system designed and developed by IBM with the collaboration of the University of Toronto, since 1998. It is focus on supporting heterogeneous architectures, thus specifically targets Shared-Memory symmetric Multi-Processor (SMMP) and NUMA 64-bit computing systems (currently running on PowerPC and Microprocessor without Interlocked Pipeline Stages (MIPS) platforms) [36]. The OS, which consists mainly in a Linux-compatible kernel, is based on the *Tornado* and *Hurricane* operating systems [37], both developed by the University of Toronto.

K42 aims at reaching the following goals [36, 35]:

- *scalability and performance*: K42 efficiently scales on a variety of heterogeneous systems, from large multi-processors and NUMA systems to small multi-processors or single-processor systems;

- *adaptability*: K42 manages system resources in a way that matches the evolving needs of the running applications, contributing to the autonomic behavior of the overall system;

- *customizability, extensibility and maintainability*: due to its open source nature, K42 guarantees a natural high degree of customizability; moreover, it is extensible in the sense that allows new platforms and applications to be simply added; last, the possibility of straightforwardly upgrade the system with new components without interrupting the services makes it easily maintainable.

All these goals are reached through simple yet interesting design principles:

- *object-oriented design*: K42 implements each system resource (i.e., an open file or a running process) as an object, storing a reference to it in a globally shared Object Translation Table (OTT). This object-orientation allowed the development of a scalable kernel, since every system resource is managed by a *per-instance* object or set of objects. This choice wisely guarantees applications the ability to best serve their needs, which can vary as the time goes on. Autonomic capabilities are provided to swap *on-the-fly* these per-instance objects;

- *avoidance of centralized code and data structures*: the K42 design includes the use of distributed code and data structures, allowing the programmer to avoid global locks, which usually degrade both performance and scalability;

- *micro-kernel design* [38]: the overall structure of K42 is based on a micro-kernel design, made up of a small exception-handling component (the micro-kernel) and many servers which marshal all of the operating system functionalities. The micro-kernel is in charge of providing basic functionalities (memory, process, and network management, IPC, ...), while servers provide more advanced OS features (file system, sockets, ...). Moreover, some system functionalities are moved from the kernel to user-space libraries, allowing applications

developers to redefine the behavior of such modules.

Figure 2.6 should help the reader to better understand the structure of the K42 OS.

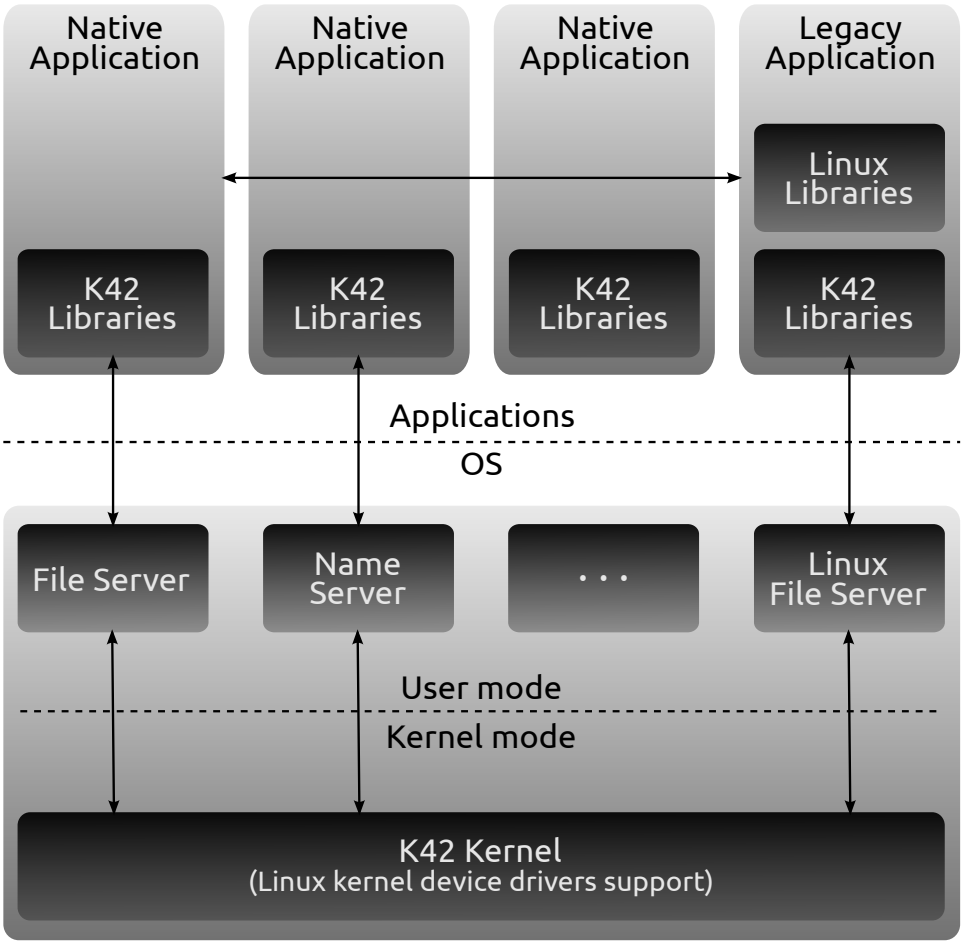

Figure 2.6: The K42 operating system structure.

Looking more specifically into the autonomic capabilities K42 is equipped with, the object oriented design allows the support of online reconfiguration [39] and dynamic update [40] mechanisms. These mechanisms allow the components of the OS to be modified on-line and to apply updates to the system without any downtime. The realization of these ideas is made possible trough the so called *hot-swap* procedure, which enables

the switch among available implementations of the same *living* component. More specifically, monitoring code, diagnostic code and implementations can be dynamically inserted and removed in functioning systems [41, 42]. Thanks to the hot-swap mechanism, K42 can be highly optimized for the common case of execution, hot-swapping ad-hoc solutions for uncommon cases. With the same idea in mind, caches and memory management policies can be changed at runtime, taking into consideration the current data access patterns in order to always use the best performing policy; shared and partitioned versions of the file pages caching mechanisms are hot-swappable, allowing optimization for sequential or highly parallel applications. Exploiting user-space libraries, applications can provide specific implementation of the OS services that can be hot-swapped by K42, if needed. Last, monitoring objects can be interposed into the relevant OS code sections by applications that benefit from the information gathered by such monitors: in this way applications that do not require monitoring are not slowed down by the execution of the monitoring code.

2.2 State of the Art

Thanks to the definitions given at the beginning of this chapter, it is possible to clearly state which are the main objectives of this thesis:

1. the implementation of a monitoring infrastructure which would allow to gather information about the level of lock-protected data contention among different threads in the system;

2. the design of a decision mechanism able to improve applications performance in term of their execution time, by mapping tasks on cores taking into consideration the information provided by the newly designed monitor;

3. actually find a way, i.e., the right actuation technique, to perform this tasks migration.

With this aims clear in mind, the state of the art related to this topics is to be investigated. In particular, the field of system monitoring and tracing (related to point 1.) and the one of self-aware tasks scheduling and mapping (according to the points 2. and 3.), with reference not only to contention but also to other relevant quantities.

2.2.1 System Monitoring

Augmenting an operating system with autonomic capabilities cannot abstract from a simple and lightweight, yet powerful and comprehensive monitoring infrastructure. The monitor must be able to provide to decision engine all the information needed to perform its work on the system, and only it: for this reason it must be carefully designed and implemented. There are many quantities their monitoring could be interesting (cores temperature and power consumption, applications performance, resources contention, ...): some of them are easily accessible and quantifiable, some others are trickier to be retrieved and synthesized.

Temperature monitoring is quite straightforward to be performed, due to the presence of specialized sensors on all modern architectures. Moving from single-core processors to multi/many-core architectures, temperature sensors evolved from a single analogical element located in the middle of the Integrated Heat Spreader (IHS) (and monitoring the temperature of the whole package) to several on-chip Digital Thermal Sensors (DTS) located in the most significant hot-spots of each processor [43]. Since many sensors are present for each core, a more accurate measurement is possible: by convention the core temperature is the highest among the measured ones. Temperature sensors resolution is usually 1 Celsius degree. The data measured by these sensors are stored in processor registers and made available

to the OS through simple interfaces, depending on the architecture producer (see [14] for an example).

The interest in monitoring also processors *power consumption* is becoming more and more important, due to the increasing diffusion of mobile battery-constrained devices. In opposition with temperature, on-line measurement of processors power consumption proved to be difficult: the simplest and most accurate way to do that is by connecting an oscilloscope to the output pins of a processor. Obviously, this is possible only in laboratory and for research purposes, but it is not feasible on common devices [44]. When dealing with mobile devices, a power consumption estimation can be performed by evaluating the variation of the battery charge level [45, 46]. In the case of desktop or server machines, mathematical models can be built [47, 48]: a model can provide only an estimation and usually its accuracy its directly proportional to its complexity.

Other two fundamental quantities to be monitored to implement autonomic capabilities are applications *performance* and resources *contention*. Both these topics proved to be really interesting and challenging: for this reason, next sections (Sections 2.2.1 and 2.2.1, respectively) are devoted to their investigation.

Performance

In this Section, performance monitoring and tracing infrastructures are investigated. In particular the ones that allow K42 and Sefos to have a performance-aware behavior are presented. A lot of attention is dedicated to a framework not introduced before: Heart Rate Monitor (HRM). This framework is based on ideas similar to the ones proposed by Application Heartbeats (the Sefos monitoring layer), but aims at improving it under many aspects. It has been developed by the Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) resource group

at Politecnico di Milano and it is the framework chosen to be modified and adapted in order to offer information about resource contention too.

K42 Monitoring and Tracing Infrastructure As described in Section 2.1.2, the K42 operating system shows an autonomic behavior: in order to retrieve enough information to do so, it is equipped with a tracing infrastructure that manages the logging of any interesting system event [49]. The coupling of such an infrastructure with the K42 ability to hot-swap system components and dynamically insert monitoring objects into the OS code, is crucial for the system to show autonomic capabilities. This framework is characterized by the following features:

- A unified set of event is available for each monitoring activity: from correctness debugging, to performance debugging and monitoring.
- It is lightweight and non-invasive in the sense that, even when not in use, the monitoring infrastructure is kept compiled-in, allowing data gathering to be dynamically enabled or disabled at runtime. However, it is also possible to exclude it from the compilation, if zero impact is to be obtained.
- As the definition of monitor explains, the infrastructure is in charge only of collecting and making available the gathered information, leaving the analysis task to another component.
- The event logging mechanism is flexible enough to provide cheap collection of data for both small and large amounts of data per event.

Sefos Monitoring Infrastructure – Application Heartbeats The monitoring infrastructure offered by SEEC for the Sefos OS is known with the name of Application Heartbeats [50]. Application Heartbeats offers a portable, simple and usable user-space library for monitoring an application actual

progress towards its goals. This framework implements a simple, yet effective and extremely powerful monitoring infrastructure: the API is made of a small set of functions that makes it straightforward to use. This API provides a simple abstraction, the *heartbeat*, which allow to measure applications performance in critical sections. More formally, an heartbeat can be defined as a periodic signal sent from the application to the API to indicate its progress. Heartbeats makes it possible to declare performance goals through another simple concept: the *heart rate*. The hear rate is simply defined as the number of heartbeat generated by an application in a time unit and is measured in $\frac{\mathrm{heartbeats}}{\mathrm{seconds}}$ $\left[\frac{\mathrm{hb}}{\mathrm{s}}\right]$.

Any application using the Application Heartbeat API has standardized methods to:

- assert its performance goals specifying a certain number of parameters when it registers: minimum and maximum heart rate, the size of the monitoring window, the size of the heartbeats history buffer, and others.
- update at runtime its progress calling a function that emits an heartbeat. The framework automatically updates all the necessary information about the global heart rate and the window heart rate, and other internal structures;
- monitor the progress of the execution. The available information is made available to either external interested observers or to the application itself: these two scenarios are shown in Figure 2.7.

A typical example of Application Heartbeats use is a video encoder [50], which measures the quality of its service (QoS) in frames per second: $30-35$ frames per second are usually to be delivered to offer a good quality. The video encoder can be instrumented in order to generate an heartbeat every time a video frame is processed. An external observer can consequently

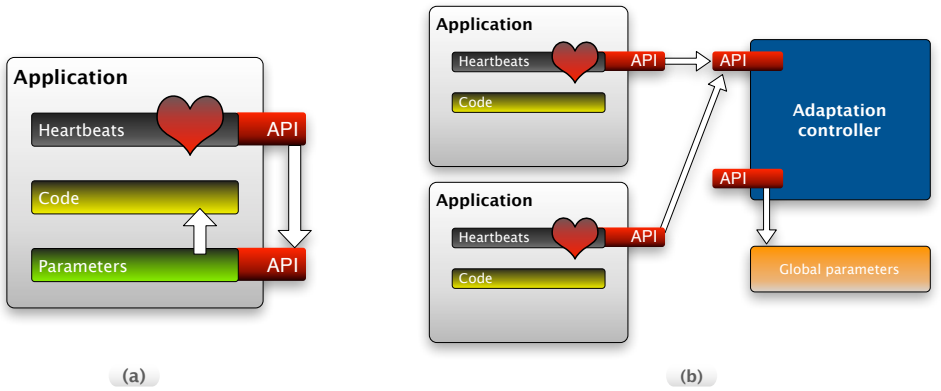

Figure 2.7: From [7] – (a) Heartbeats statistics used by the application itself to perform self-optimization; (b) optimization of system parameters by an external observer, working on one or more applications.

improve (or reduce) the encoder performance through the modification of some parameters, such as the number of cores assigned to it.

The Application Heartbeats monitor is particularly suitable for instrumenting applications where there is a computational intensive code section, realized as a loop in the code: one heartbeat is sent at each iteration of the loop. Unfortunately, it is not possible to find such a behavior in all the applications: Application Heartbeats proved to be a lightweight and effective monitoring infrastructure, even if limited to a specific class of applications [51, 52, 53].

Heart Rate Monitor (HRM) The ideas behind Heart Rate Monitor (HRM) [54] resemble those behind Application Heartbeats (see the previous Section). However, the authors aim at improving its functionality. The fact that Application Heartbeats is a portable user-space active monitor prevents a portion of commodity operating systems (i.e., the kernel) to easily access the information it provides, making the development of kernel-space adaptation policies troublesome. Moreover, Application Heartbeats only supports multi-threaded applications forgetting about multi-processed appli-

cations and makes use of synchronization even for signaling progresses. HRM is an active monitor, integrated with Linux, supporting applications with multiple threads, multiple processes, and any feasible mix of threads and processes, which avoids synchronization to reduce its overhead as much as possible. HRM sacrifices portability to functionality and, just like Application Heartbeats, it exposes a compact API, allowing applications and system developers to instrument applications and build both user- and kernel-space adaptation policies. The HRM framework is deeply analyzed in this Section, since it represents the starting point for the implementation of this research work.

The HRM framework inherits from Application Heartbeats the definition of *heartbeat* as a signal emitted by any of the application's tasks at a certain point in the code and indicating application's progresses. A new concept is introduced: *hotspot*; an hotspot is a performance-relevant portion of code executed by any of the application's tasks and usually abstracts the most time consuming portion of a program. Since an application is a set of tasks pursuing a set of objectives, any of the tasks working towards one of such objectives can emit heartbeats. For this reason, the definition of *group* as a subset of application's tasks pursuing a common objective was introduced. Groups are non-intersecting subsets; hence, a task belongs to only one group at a time. It is important to notice how such a definition does not neglect the existence of multi-grouped applications (e.g., a group encoding the audio stream and a group encoding the video stream in an audio/video encoder), a case Application Heartbeats completely neglects. The concept of group allows HRM to support multi-programmed applications adopting multiple threads, multiple processes, or a mix of both processes and threads: it is enough to attach each of the application's tasks to the relevant group. Within HRM, a unique Group IDentifier (GID) identifies a group. Given the definitions of hotspot and group, it comes natural to define a re-

lation n to 1 between such entities. Each of the tasks belonging to a group executes the same hotspot, which is characterized by its heartbeats count, performance measures, and performance goal. The *heartbeats count* is linked to the number of times each task executed the hotspot. *Performance measures* are expressed in heartbeats per second and capture the concept of heart rate, which is the frequency at which tasks emit heartbeats. The *performance goal* is expressed as a desired heart rate range, delimited by a *minimum heart rate* and a *maximum heart rate*, similarly to Application Heartbeats.

The implementation of HRM consists of two parts, a user-space library and the kernel-space code. The user-space library exposes the API for both applications and systems developers; the API's basics are reported in Table 2.1. While the API's functions for applications developers grant the ability to instrument applications, providing a way to specify performance goals and signal progresses, the API's functions for systems developers are meant to retrieve applications' performance measures and performance goals.

The API exposes two functions, `hrm_attach` and `hrm_detach`, to attach the current task to the group identified by a GID and to detach the current task. Two functions, `hrm_set_active` and `hrm_set_inactive`, are implemented to either set active or inactive the current task: a task is said to be active if it is executing the hotspot, inactive otherwise. These two states prove to be useful to maintain performance measures in programs using "spawn & kill" parallelization (e.g., x264 in the PARSEC 2.1 benchmark suite [55]), in which there is no guarantee that at least one active task is always alive.

Different applications may be concerned with either long- or short-term trends. Therefore, the API exposes both `hrm_get_global_rate`, to catch long-term trends through the average heart rate over the whole execution time, and `hrm_get_window_heart_rate`, to catch short-term trends (i.e.,

Function	Description
`heartbeat`	Emit a heartbeat
`heartbeatN`	Emit n heartbeats
`hrm_attach`	Attach the task to group identified by GID
`hrm_detach`	Detach from group
`hrm_set_{active|inactive}`	Set the task active or inactive
`hrm_set_{min|max}_heart_rate`	Set the minimum or maximum heart rate
`hrm_set_window_size`	Set the sliding window size
`hrm_set_timer_period`	Set the timer period
`hrm_get_{global|window}_heart_rate`	Get the global or window heart rate
`hrm_get_{min|max}_heart_rate`	Get the minimum or maximum heart rate
`hrm_get_{window_size|timer_period}`	Get the window size or the timer period

Table 2.1: Functions exposed by the HRM user-space API

variable-length trends) through the heart rate measured over a time window. The window size, which is expressed in timer periods, is used to control the amount of past measures to account for; the timer period controls how often performance measures are updated. The window size and the timer period can be set through `hrm_set_window_size` and `hrm_set_timer_period` respectively. Two additional functions, `hrm_set_min_heart_rate` and `hrm_set_max_heart_rate`, are exposed to adjust performance goals, which are defined as a desired heart rate range. Other functions are available to retrieve performance goals and performance goals related parameters. The most important API's functions are `heartbeat` and `heartbeatN`. Calls to these functions are inserted within the hotspot of a program to signal progresses by incrementing the summation of heartbeats either by 1 or by a generic integer value.

The kernel-space implementation of HRM consists of an API that mimics a subset of the functions described above, and the core of the active

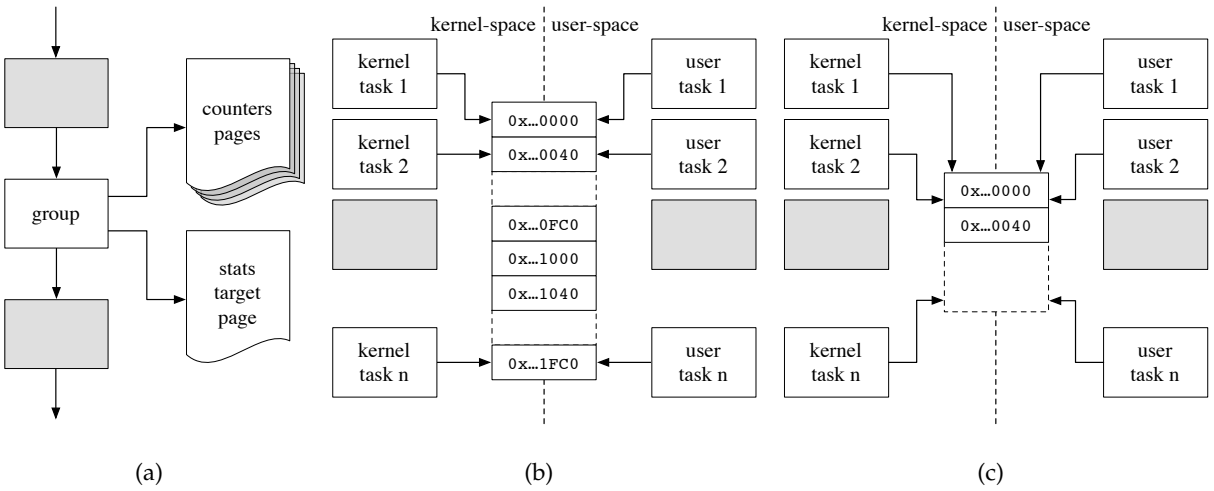

Figure 2.8: Figure 2.8(a) denotes the structure of the implementation of HRM; Figure 2.8(b) shows the organization of memory pages devoted to heartbeats count; Figure 2.8(c) shows the organization of the memory page dedicated to performance measures and performance goal

monitor. Figure 2.8(a) shows the globally accessible list of groups at the very base of the implementation of HRM. The list of groups can be read in parallel and written serially by `hrm_attach` and `hrm_detach` function calls; to guarantee correctness, the list of groups is protected by a read-/write lock. Each group is provided with a set of memory pages devoted to heartbeats count and a memory page dedicated to performance measures and performance goal. The amount of memory pages to store heartbeats is a compile time tunable parameter.

Memory pages are shared between the kernel-space and the user-space to reduce the overhead in accessing the information as much as possible. More specifically, the content of memory pages devoted to heartbeats count is the most critical to HRM since it can be concurrently accessed at a high rate by both kernel-space tasks and user-space tasks. A way to avoid overheads and concurrency issues consists in splitting the heartbeats count in a set of per-task heartbeats counts; hence, function calls to both `heartbeat`

and `heartbeatN` reduce to an atomic variable increment. The amount of heartbeats counts stockpiled in memory pages is architecture dependent since they are cache line aligned. The implementation of HRM instantiates standard-sized memory pages of 4 Kbytes and x86 and x86-64 microprocessors feature cache lines of 64 bytes: this implies that each memory page can contain up to 64 heartbeats counts. Figure 2.8(b) shows the organization of the memory pages devoted to heartbeats count focusing on tasks accessing dedicated cache line aligned heartbeats counts.

Different applications and adaptation policies may be concerned with either long- or short-term trends. Therefore, the 64 bytes of the memory page dedicated to performance measures and performance goal contain both a global heart rate, which accounts for the whole execution of a group and catches long-term trends, and a window heart rate, which accounts for the execution of a group over a time window and catches short-term trends. The global heart rate and the window heart rate are respectively computed according to Equation 2.1 and Equation 2.2. In the Equations, g indicates the group, t indicates the current time, t_0 indicates the time at which the group was created, and t_w indicates the time at which the window started. The performance measures are asynchronously updated in kernel-space in the context of a High-Resolution (HR) timer.

$$ghr_g(t) = \frac{\sum_i cnt_i(t)}{t - t_0} \tag{2.1}$$

$$whr_g(t) = \frac{\sum_i cnt_i(t) - cnt_i(t_w)}{t - t_w} \tag{2.2}$$

The second chunk of 64 bytes of the memory page, dedicated to performance measures and performance goals, contains a minimum heart rate and a maximum heart rate to define a performance goal through a heart rate range. Other available parameters are the window size and the timer period; the latter sets the frequency at which performance measures are updated, while the former sets the window size expressed in timer peri-

ods. Figure 2.8(c) shows the organization of the memory page dedicated to performance measures and performance goal; each task accessing these information maps the whole memory page.

Resource contention

With reference to the work described in this dissertation, the most interesting quantity to be monitored is the contention of resources among different threads. Monitoring such a quantity is not that easy as it could be for temperature: there are no sensors able to measure how much a data is contended among different tasks. Previous works about this topic faced this issue mainly undertaking two different roads: by exploiting hardware techniques or by implementing higher-level software approaches.

Hardware Approaches Detecting sharing patterns of threads automatically has always been a challenge. One of the first idea in this direction was to exploit page protection mechanisms to identify active sharing among threads. In [56] this approach is used to implement software Distributed Shared Memory (DMS). However it has some important drawbacks [57]: first of all, the coarse granularity of detecting page-level contention can lead to an high degree of false sharing. Moreover, protecting pages results in high overhead with an attendant increase in page-table traversals and Translation Look-aside Buffer (TLB) flushing operations.

A more effective and low-weight way to gather data about contention is by using the data sampling features of the Performance Monitoring Unit (PMU) available in today's processing units. PMUs integrate Hardware Performance Counters (HPCs) that can be used to monitor and analyze performance in real-time, offering finer-grained information and having far lower overheads (since most of the monitoring is offloaded to the hardware) [57]. HPCs allow the counting of detailed micro-architectural events

in the processor, such as branch miss-predictions and cache misses. Thresholds can be set on individual quantities in order to interrupt the processor when they are overcome. Moreover, PMUs make additional registers available for user inspection: from the addresses that cause cache misses to the corresponding offending instructions.

The quantities that can be monitored through HPCs are not directly related to data contention. However programmers often use these counters to improve application performance by monitoring section of code to detect and optimize hotspots and by building a model from this information in order to quantify the contention among different tasks. For example in [57] the system maintains vector of cache accesses to detect threads that share data and examines memory access stalls to determine which threads are using data from a faraway cache. A different methodology, always exploiting HPCs, is described in [6]: the *marginal gain* metric is defined as the derivative of task's miss-ratio curve over time. Hardware counters are used to measure the miss-ratio and some modifications to cache controllers are introduced. The observation subsystem proposed in [58] inspects relevant performance counters, gathering information on a per-thread basis. In particular, processors counters taken into account during measurements are: Last Level Cache (LLC) misses, LLC references, instruction retired, core cycles, and reference cycles. The choice of the performance counters to be taken into consideration while designing a monitoring system exploiting them is fundamental. In fact, the quantity of counters that can be enabled is limited. Moreover many constraints are posed on their use and the documentation describing them is quite poor [57].

A great work on contention monitoring was performed by Federova *et al.* [59, 60, 4]. After a deep state of the art analysis, the authors came up with a methodology which allows to identify the *solo LLC miss rate* as one of the most accurate predictors of the degree to which applications

will suffer when co-scheduled. Then, they propose the design of a threads classification scheme, according to their memory behavior, used to design a suitable scheduling policy.

Software approaches The resource contention monitoring issue is traditionally related to the use of hardware counters: only a few examples of meaningful software approaches relying on system simulation can be found in literature. Among them the most important proved to be the one by Cho and Jin [61]: this work involves simulation on a chip multithreading and multiprocessing processor, comparing private and shared caches among cores in a platform with a novel architecture in which memory pages map into *cache slices*. Exploiting this enhancement, the authors demonstrated that the OS, using its knowledge of memory page usage, can make intelligent cache management decisions.

The monitoring infrastructure proposed in Chapter 3 and described in Chapter 4 does neither exploits hardware counters (due to the limitations previously exposed) nor architecture simulation. The designed approach aims at instrumenting a user-space lock library with the HRM framework (presented in 2.2.1), exploiting it not to monitor application performance, but lock contention among threads.

2.2.2 *-aware Scheduling and Mapping

Autonomic capabilities can be shown in many different ways within an operating system: by dynamically swapping the implementation of a system feature with reference to the present conditions [23, 7], by adjusting cores frequencies in order to meet security requirements related to temperature [62, 63], by modifying the applications nice value to meet their goals [7]. This work of thesis is focused on introducing autonomic capabilities in a system by working on how tasks are mapped on the available

cores, with the explicit aim of reducing resource contention. For this reason, this Section investigates the state of the art on scheduling and mapping algorithms which take into consideration several run-time information to perform their job.

Power-aware and temperature-aware scheduling are both interesting and challenging topics, but unfortunately go beyond the scope of this dissertation. The former topic is well introduced in [53], which proposes *PowerDial* a system for dynamically adapting application behavior to execute successfully in the face of load and power fluctuations, actually reducing the computational resources that the application requires to produce its results. These reductions translate directly into performance improvements and power savings. Experimental results show also that PowerDial can significantly reduce the number of machines required to service intermittent load spikes, enabling reductions in power and capital costs.
A comprehensive overview on temperature-aware scheduling is given in [64]. Moreover, the authors propose *Dimetrodon*, a framework implementing a software preventive thermal management mechanism by the injection of CPU idle cycles while scheduling tasks. This technique proved to be extremely flexible and demonstrated its efficiency compared to hardware techniques under throughput and latency-sensitive real-world workloads, achieving profitable trade-offs for temperature reductions up to 30% due to rapid heat dissipation during short idle intervals.

Performance-aware and contention-aware scheduling are, instead, strictly related to the work proposed in the next chapters. Thus, both of them are deeply analyzed in the following sections. First of all, however, a brief introduction to the evolution of the process scheduler in Linux is necessary: the Linux scheduler does not provide an autonomic behavior, but represents the base case on which some of the presented works have been built and this thesis work is built on, as well.

Base case: the Linux Scheduler

One of the components that is most frequently updated and improved in an operating system is the process scheduler. The Linux process scheduler does not represents an exception: this section analyses the evolution of the scheduling algorithm from the its first implementation to the present Completely Fair Scheduler (CFS). This component has been continuously improved, thanks to, for example, load balancing techniques, but it is not possible to state that it shows autonomic capabilities: many improvements are thus possible under this point of view.

The scheduler algorithm included in the first versions of the Linux kernel (released by Linus Torvalds in 1991 [8]) did not aim at obtaining the best performance, but above all to be reliable. It was very simply designed, providing quite poor scaling capabilities with reference both to the number of executing processes and to the number of available execution units. Despite this, the first implementation of the algorithm did not change until 2004, when Linux 2.6.8.1 was released [65]. In this Linux release, a new version of the scheduling algorithm was inserted: it was designed in order to take into account the heterogeneous targets of Linux (used both in desktop and in server environments). The main feature of this new scheduler, the one that gives the name to it, is its ability to pick the next task to be executed in a constant time, not depending on the number of tasks to be scheduled or on the number of available processors. Using a mathematical term, it has a $O(1)$ complexity, with the Big-O notation [66]. This important feature allows the process scheduler to achieve better scaling performance than the previous implementation, introducing two fundamental data structures:

- a *runqueue* structure for each processor in the system, storing a pointer to the task assigned to that CPU for execution;
- two *priority arrays* for each runqueue: tasks are moved from the first

one, named *active* to the second one, named *expired*, when they run out of their execution quantum. When the active array is empty, the scheduler simply inverts the two labels. The two priority arrays contain linked lists, one for each priority level.

Each task is assigned a priority level by combining a static value, called *nice*, and a dynamic value computed by the scheduler to improve interactivity. In the O(1) scheduler, this is possible through a quite complex heuristics (well describe in [67]) which classifies the task into CPU-bound or I/O-bound, giving the latter ones higher priority.
This implementation of the scheduling algorithm introduces, for the first time in Linux, the idea of load balancing: in order to keep the workload distributed on the available processors, tasks are moved from overloaded runqueues to the underloaded ones.

A great innovation in the scheduling algorithm was introduced by the Completely Fair Scheduler (CFS), released with Linux 2.6.23 in 2007 [8], and representing the scheduling algorithm used in the current versions of the kernel. This scheduler was designed by the same author of the O(1) scheduler and aims at solving the limitations of it predecessor: the complex heuristics for tasks classification mainly. To do that the basic idea of Rotating Staircase Deadline Scheduler (RDSL) was embraced: being fair in CPUs assignment without trying to characterize the behaviour of each task [68]. The fairness the name of the scheduler refers to is related to assigning computing resources to the tasks in execution. This is obtained by introducing the concept of *virtual runtime* and by discarding the concept of quantum of execution, fundamental in the O(1) scheduler implementation. The *virtual runtime* [8] of a task is the actual amount of time spent by the task running on a processor, normalized by the number of runnable tasks. This quantity is measured in nanosecond and represents the time the task would have run on an ideal machine able to support perfect parallel execu-

tion, with an ideal number of processors equal to the number of runnable tasks. The CFS updates the virtual runtime of each task periodically (at each kernel tick) and, when needed, chooses the task with the lowest value of virtual runtime for execution.
The CFS algorithm inherits the runqueues data structure from the $O(1)$ scheduler, but implements it in a different way. In order to optimize performance in choosing the next task for execution, runqueues are implemented as *red-black trees* [66]. This particular class of balanced binary trees allows insertion and deletion of elements with a $O(\log(n))$ complexity, where n refers to the number of tasks in the tree, and is topologically ordered so that the task with the minimum virtual runtime is always the leftmost leaf of the tree [8]. The algorithm previously described achieves good scheduling performance with reference both to interactivity and throughput, while it does not really allow the user to influence the scheduler job.
In CFS the load balancer was improved too: with the diffusion of multi-core architectures its role become crucial. The load balancer is invoked periodically by the scheduler code, randomly on one of the CPUs available in the system. The balancing algorithm tests whether there is in the system a core that is busier than the one on which it is executing. If the answer is yes, the busiest runqueue is selected and one task is moved from that runqueue to the current one. On the other hand, if the answer is no, nothing happens. This algorithm relies on the fact that its code will be statistically executed with the same frequency on each CPU. This is true, in the sense that experimental results show the load balancer allows, on average, to reach an even fairer distribution of the tasks on the CPUs runqueue.

Performance-aware Scheduling and Mapping

Once the most interesting infrastructures for performance monitoring has been introduced, it is time to see how the information gathered by them

is exploit in order to make the scheduling algorithm autonomic and behave differently according to the changing conditions.

Scheduling in Sefos Due to the fact that the SEEC framework is completely implemented in user-space, the Linux scheduler algorithm was not directly modified, but its decisions are indirectly affected from a user-space library [19]. More specifically, the work of the scheduler in SEEC is guided by a service named *core allocator*. This service is in charge of assigning a subset of the available processing units in the system to the running applications. This is possible, in Linux and from user-space, by duly modifying the *affinity mask* of the interested process. An affinity mask is a bit mask associated with each task indicating what processor(s) it should be run on by the scheduler of the OS. Each bit in the bit mask represents an available processor: if the corresponding bit value is 1 the task is allowed to run on that processor, if it is set to 0 this is not true.

Experimental results [69] show that this approach is enough to obtain performance improvements. However the realization of such service in user-space poses a number of limitations on the precision of the service in affecting the system status. The main limitation is that the core allocator can only indirectly map the tasks on the available processors, by modifying their affinity mask, but the final decisions are taken by the kernel-level Linux scheduler, which autonomously determines, for instance, *when* a certain task is to be moved. Despite the simplicity of the approach, this main drawback convinced the authors that a kernel-space implementation of the same service would be advisable when merging the SEEC framework with the fos operating system.

Scheduling in K42 The tracing infrastructure that K42 offers, allows the implementation of a performance-aware scheduler. This component implements an interesting approach, known as *two-level scheduling* [70], which is

designed in order to divide the process scheduler in two subsystems: the first one running in kernel-space and the second one in user-space. This approach, absolutely consistent with the micro-kernel idea the OS is based on, permits to achieve the following advantages:

- Improve performance by having only a single thread scheduler running in user-space (which avoids context switches), without the kernel ever being aware of what it does;
- Allow applications to tailor the scheduler at the user-space level according to their needs, simply by reimplementing the library.

The user-space scheduler is in charge of managing threads belonging to the same process (thus, sharing the same address space). Threads are packed into an entity called *dispatcher*: the kernel-space scheduler is not able to distinguish among threads contained in the same dispatcher, leaving the burden of managing them to the user-space scheduler. The kernel-space scheduler can only schedule the dispatchers, assigning them the resources by using *resource domains*, entities at a higher level in the hierarchy. More formally, each resource domain groups a set of dispatchers; the rights to use the hardware resources is given by the kernel-space scheduler to a resource domain. Resource domains are supposed 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.

Such an approach leaves a lot of freedom and control to the application developer, relying on its ability. In fact, the programmer could decide to create a process using a single dispatcher with many threads and define the scheduling policy that marshals the threads by reimplementing the user-space scheduler code. On the other hand, the programmer could use multiple dispatchers to obtain real parallelism or to assign them different

scheduling characteristics.

Performance-Aware Fair Scheduler (PAFS) As described in Section 2.2.1, the HRM framework has been developed within the Linux kernel. Thus, the Performance-Aware Fair Scheduler (PAFS) [54] modifies, exploiting data coming from HRM, the Linux Completely Fair Scheduler (CFS) in order to make it aware of the applications performance. When designing PAFS the authors considered three main goals: first, applying a *best-effort* approach to drive the instrumented applications towards meeting their performance goals; second, being able to flawlessly manage also legacy applications; and third, being safe, ensuring that no task of any application (either instrumented or legacy) ever results in *starvation*. PAFS observes the performance measures of the instrumented applications, trying to speed up or slow down their tasks according to whether they are matching or not their performance goals.

HRM allows expressing performance goals in terms of a heart rate range delimited by a minimum heart rate and a maximum heart rate. These two bounds are interpreted by the proposed adaptive process scheduler as follow: the minimum heart rate defines a strong lower bound for performance while the maximum heart rate defines a weak upper bound for performance. The adaptive process scheduler assigns microprocessors' time to tasks in order to keep the performance measures of monitored applications above their minimum heart rate, penalizing as needed the ones that are performing over their maximum heart rate. However, no guarantees on performance goals matching are given (for instance, this could even be impossible due to resources scarcity).

The default, performance-unaware CFS scheduler implemented in Linux exposes two interesting properties that make it a solid base to build PAFS on: *fairness* and second *non-starvation*. PAFS introduces the concept of per-

formance-aware fairness, meaning that the process scheduler gets fair in assigning microprocessors time accounting also for applications performance and performance goals. The introduction of performance-aware fairness consists in modifying the computation of the virtual runtime, taking into consideration application's current performance and its performance goals. The non-starvation property is proved to be preserved by these modifications. The decision policy which decides how much the application' current performance and its goals are related to the virtual runtime is a simple yet effective heuristics, which defines this relationship as the ratio between either the global heart rate or the window heart rate and an average between the minimum heart rate and maximum heart rate. In this way, tasks are progressively advantaged when their heart rate is less than the minimum heart rate while they are progressively disadvantaged when their heart rate is greater than the maximum heart rate. When the heart rate matches the performance goal the behavior of PAFS replicates that of CFS.

Other interesting research works related to performance-aware scheduling are Performance-Driven Processor Allocation (PDPA) [71] and Scheduler for Multimedia And Real-Time applications (SMART) [72, 73]. The former project focuses on processor allocation in shared-memory multiprocessor systems, where no knowledge of the application is available when applications are submitted. *SelfAnalyzer* is used to dynamically analyzing speed-up, efficiency and execution time of running applications and a new scheduling policy that distributes processors considering both the global conditions of the system and the particular characteristics of running applications is designed. The importance of the interaction between the medium-term and the long-term scheduler to control the multiprogramming level in the case of the performance-aware scheduling policies is also highlighted. The second project, SMART, supports applications with time constraints, and provides dynamic feedback to applications to allow them to adapt

to the current load. The integrated support for real-time applications and conventional processes allows the user to prioritize across real-time and conventional computations, and dictates how the processor is to be shared among applications of the same priority. Dynamic and seamless resource allocation is also granted by the framework itself: real-time tasks are shed and their execution rates are regulated when the system is overloaded, while providing better value in underloaded conditions than previously proposed schemes.

These projects are not further investigated in this document (the interested reader can refer to the cited bibliography for more information) in order to focus the attention on the state of the art on contention-aware scheduling (treated in the next section).

Contention-aware Scheduling and Mapping

While the contention monitoring approaches proposed in literature differ a lot from the one proposed in this work (based on HRM, as explained in Chapter 3), the scheduling policies that have been designed provide an interesting background to compare and improve the one presented in this document.

DI, DIO, and DINO Based on the classification schemes derived from the evaluation of the LLC miss rate (via hardware performance counters), a scheduling policy named Distributed Intensity (DI) was designed [59]. In the DI algorithm all the tasks are assigned a value equal to their solo miss rate, and classified as memory *intensive* or *non-intensive*. The goal is then to spread the threads across the system such that the miss rate are distributed as evenly as possible. In order to move the tasks in the best way possible, the notion of memory hierarchy entities is taken into account. *Memory hierarchy entities* are distinct hardware modules (e.g., cores, chips, packages)

each of which is located on its own level of memory hierarchy (see Section 1.2.3). During the initialization phase, the algorithm determines the number of memory hierarchy levels and the number of distinct entities on each level. DI then tries to even out the miss rate on all levels of memory hierarchy, assigning tasks on the base on the solo miss rates of the applications. The real miss rate of applications will change when they share a cache with a co-runner. The DI scheduler is implemented as a user-space scheduler running on top of the Linux kernel. It enforces all scheduling decisions via system calls which allow it to bind threads to cores.

However, DI uses solo miss rate estimated using stack distance profiles as the input to the classification scheme. The stack distance profiles require extra work to obtain while the algorithm is running, thus the presented approach is not feasible online. An improvement of such algorithm, exploiting the same classification scheme and scheduling policies as DI, was implemented in order to obtain the miss rates of applications dynamically online via performance counters. This algorithm is named Distributed Intensity Online (DIO) [59, 60]. The dynamic nature of the obtained miss rates makes DIO more resilient to applications that have a change in the miss rate due to LLC contention. While running, DIO continuously monitors the miss rate of applications and thus accounts for phase changes. To minimize migrations due to phase changes of applications, miss rates are collected not more frequently than once every billion cycles and an average of them is used for scheduling decisions. Every billion cycles the new miss rates are measured and the tasks assignment is re-evaluated based on the updated miss rate running average values for the workload. As DI, also DIO is completely implemented in user-space, thus managing the assignment of tasks to cores using affinity interfaces exposed by the Linux kernel.
The DIO algorithm proved to have the inherent ability to predict when a group of tasks co-scheduled on the same memory domain will improve

or degrade each other's performance. Further research revealed that this ability could be exploited to build a power-aware scheduler (called DIO-POWER) that would not only mitigate resource contention, but also reduce system energy consumption. The idea which inspired the design of such algorithm is that clustering tasks on as few memory domains as possible reduces power consumption [59].

The same research group tried to bring the same algorithm on a NUMA system obtaining poor results [74]: not only contention was not manage efficiently, but sometimes performance were even hurt when compared to a default contention-unaware scheduler. This is due to the fact that NUMA-agnostic migrations fail to eliminate contention for some of the key hardware resources on multi-core systems and create contention for additional resources. To overcome these difficulties a new version of the DIO algorithm was designed and named Distributed Intensity NUMA Online (DINO). DINO prevents superfluous thread migrations, but when it does perform migrations, it moves the memory of the threads along with the threads themselves. DINO represents an evolution of DIO, thus it inherits the same basilar concepts, reviewed, when needed, in order to adapt to NUMA architectures.

Futex Aware Scheduling Technique A work even more related to the one presented in the next chapters is the one discussed in [75]. The aim of the Futex Aware Scheduling Technique (FAST) project is to efficiently reuse the thread's state that is already in a processor's cache by enforcing an affinity between the processor and threads executing on them, applying this idea to locks and data in critical sections protected by these locks.

The whole framework is developed, again, in user-space, modifying only partially the $O(1)$ scheduler implementation in the Linux 2.6 kernel. Contention is monitored through *Perfmon* [76], a performance monitoring

tool which allows to collect counts or samples from unmodified binaries and uses hardware performance counters. The additional information coming from this monitor enables the scheduler to take intelligent decisions for tasks that are contending for locks. In order to obtain such an intelligent behavior, the OS has been modified as follows:

- a new entry was added to the structure describing each sigle task, named `cpu_lock`. This new field is supposed to store the identifier of the physical Central Processing Unit (CPU) the task will need to run on when it acquires a lock;
- the `futex_wake()` function was modified in order to set the `cpu_lock` field for the acquiring task equal to the identifier of the CPU of the releasing task;
- the O(1) scheduler was modified to check the `cpu_lock` field when migrating a task or when trying to activate a blocked task. In particular, if the `cpu_lock` value is valid the task is accordingly migrated, if not the scheduler performs its operations as usual.

Moreover, in order to avoid any load balancing problem, the default scheduler tasks migration mechanisms is given higher priority over the presented policy.

The results reported in [75] are very interesting under the point of view of cache miss rate reduction, showing good improvements both with micro-benchmarks and with more general benchmark. However, no results are described about the overall execution time, useful to understand if a real improvement was reached. Moreover, the described approach is nowadays outdated, due to the new Completely Fair Scheduler implemented in the Linux kernel.

The two projects presented in this section are the ones that are more strictly related with and that mainly inspired the approach described in the

next chapters. Some other interesting works on contention-aware scheduling are [58] and [57]. They do not introduce original elements, thus they did not deserve a specific paragraph in this chapter.

2.3 Summary

This chapter provided a wide and high-level bird's eye view on the context this thesis deals with. First, in Section 2.1 the design principles for autonomic systems to be implemented are shown and most important operating systems providing autonomic capabilities were presented. After this introduction the work state of the art on system monitoring and *-aware process scheduling is investigated (Section 2.2). In particular, Section 2.2.1 described monitoring techniques focusing on performance monitoring (specifically HRM) and resource contention monitoring, in order to make a comparison between existing approaches and the proposed one possible. Then, Section 2.2.2 explained how the information gathered by the monitor infrastructure is exploited to improve the process scheduling mechanism, both for performance and resource contention quantities. Moreover, a brief evolution of the Linux scheduling algorithm was sketched in order to introduce the basic concepts used by the *-aware algorithms and by the one implemented for this research work.

At this point, all the concepts needed to understand the work have been introduced and the related works have been investigated. The original part of this thesis can presented: next Chapter 3 describes in the details how the monitoring framework and the scheduling policy have been designed, leaving the implementation details explanation in Chapter 4.

Chapter 3

Proposed Approach

The aim of this third chapter is to introduce the real contribution of this thesis to the research in the field of autonomic computing. The work developed in the last months is only a small brick in a more ambitious research project: CHANGE. The goal of this project is to implement a new autonomic operating system, named AcOS, from scratch: nowadays the whole system has been designed, while the actual implementation relies on the GNU/Linux operating system.

The author gave his contribution to the design of the overall idea behind the creation of the AcOS operating system and actually contributed, with the work described in this document, in the following two directions:

- in the implementation, based on the HRM framework, of a new type of monitor for lock contention among threads;
- in the design of an adaptation policy able to exploit the information gathered from the newly introduced monitor to map the tasks executing in the system to the available CPUs in order to achieve a performance improvement, in term of a reduction of the tasks execution time.

The structure and the ideas the AcOS is based on, are presented in Sec-

tion 3.1. The following two sections are devoted at explaining in the details the concepts this specific work is built on. First, in Section 3.2 why and how HRM, which is born as a performance monitor, can be converted to a resource contention monitor is shown and the modification needed to the framework in order to allow this new use are presented. Then, Section 3.3 explains the theoretical foundations that guided the implementation of an adaptation policy for the mitigation of the lock contention and the resulting improvement in tasks performance through the reduction of their execution time.

3.1 The CHANGE view

The CHANGE (standing for Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments) research group works at the Dipartimento Elettronica e Informazione (DEI) in Politecnico di Milano in the field of operating systems. The aim of the research developed by the group is the design and the implementation of a completely new operating system showing self-aware capabilities. The Operating System (OS) should be able to run on any modern computing device: from desktop to server computing systems, from mobile phones to tablets. This is possible thanks to a powerful autonomic layer that allows the system to understand the features and the limitation of the environment it is running on, continuously ensuring optimal performance. In this context the concept of *performance* is wider than usual. In fact, this term does not refers only to the throughput of the system or to the execution time of the running applications, but considers also aspects such as the system safety (in term of temperature, for example) or the system uptime and duration (considering its power consumption in relation with the battery length, in a mobile system). Thus, the operating system has to meet two types of needs:

- *system goals*: referring to objectives related to the system as a whole, such as keeping the temperature of the cores under a certain threshold in order to avoid protection mechanisms to be activated or minimizing the power consumption of the system to let it last/consume as little as possible. System goals are specified by the system designer and are strictly related to the hardware the system is equipped with.

- *application-specific goals*: are specified, through a standardized interface, by the applications running on the system. The system itself is in charge of providing the best effort in order to achieve the goals of al the applications, being fair in doing this and paying attention to its goals too.

3.1.1 Terminology

Before getting into the details of how the envisioned operating system works, some terminology is needed to avoid any kind of misunderstanding.

- *Application*. An application is an element capable of making one or more entities of the system aware of its performance goals and its current status.

- *Monitor*. A monitor is an entity equipped with sensors able to gather information from the monitored applications or from the system. Within this context, it is important to notice that goals, expressed by an application, are defined using data that can be measured through a monitor.

- *Adaptation Policy*. An adaptation policy is an element whose objective is to observe applications through monitors, in order to decide on a strategy to change the behavior of the self-adaptive computing

system for meeting the goals declared by the applications. A self-optimizing application is a special application in which the roles of application and adaptation policy co-exist.

- *Adaptation Manager*. The adaptation manager is a *singleton* element detecting whole system problems and applications meeting or not their goals, allocating (de-allocating) them to (from) adaptation policies.

3.1.2 The AcOS self-adaptive control loop

As described in Section 1.2.1, a computing system that aims at showing an autonomic behavior has to implement the so called *self-adaptation control loop*. Among the ones presented in the cited section, the control loop design chosen to be embedded in the new OS is the Observe Decide Act (ODA) control loop. Differently from the autonomic systems analyzed in the state of the art (Section 2.1), the OS here described aims at exploiting the control loop at different levels. At a lower level, the single component can benefit from autonomic management via internal ODA loop and, at higher level, a broader control loop, having a clear knowledge of all the components the system is made up of and aware of the system status as a whole, can orchestrate the different components in order to achieve system and application-specific goals. This self-awareness should allow the burden of the system parameters tuning process to be taken away from the programmer. Moreover, it should also allow the applications developers to concentrate on what their applications must do, leaving all the architecture-dependant details to be managed by the autonomic features of the system where their software will be deployed. The resulting system is a self-adaptive computing system whose structure is presented in Figure 3.1.

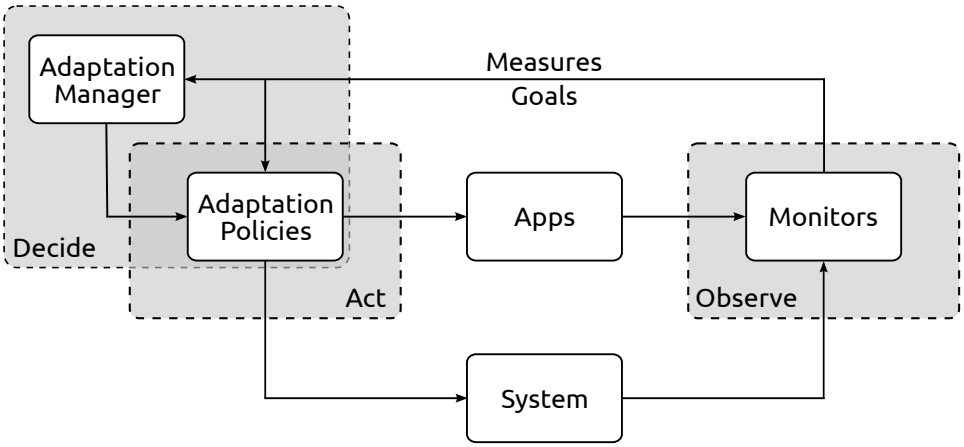

Figure 3.1: Block diagram of the approach proposed to design and implement self-adaptive computing systems.

Observe

In this context, two distinct roles are clearly defined: applications developers and systems developers. The former are in charge of writing applications and, if needed, of instrumenting them in order to provide as much information as possible to the self-adaptive computing system (e.g., user-specified goals). The latter are in charge of writing monitors, which can be either *active* or *passive* and allow the self-adaptive computing system to collect as much information as possible, adaptation policies, providing as many ways as possible to change the behavior of the self-adaptive computing system (e.g., a specialized adaptive process scheduler), and the adaptation manager. With reference to monitors, a monitor is said to be *active* when applications require to be to manually instrumented to provide information: this is the case of Heart Rate Monitor (HRM) when used as a performance monitor as described in Section 2.2.1. On the other hand, a monitor is considered passive when no intervention on the application code is required to the programmer in order to provide relevant information: a temperature monitor, exploiting cores sensors, is a passive monitor. A special case is represented by the contention monitor described in the next section:

while exploiting HRM to emit heartbeats, there is a locking library level between the monitor and the application. In this sense, the monitoring of the locking library is actually active, since its instrumentation was needed, while the monitoring of the application can be considered to be passive: if the programmer is using the instrumented locking library to manage contention in its code, no further modifications to the application are needed for contention information to be made available to the system.

Decide and Act

The resulting self-adaptive computing system exploits the ODA control loop, as pointed out in Figure 3.1. Monitors are responsible for yielding measures and goals (e.g., performance, temperature, power consumption, resource contention, ...), hence implementing the observe phase of the ODA control loop. The decide phase of the ODA control loop is partitioned between the adaptation manager and adaptation policies. The adaptation manager is in charge of detecting problems through measures and goals reported by monitors and of deciding on a strategy to allocate applications to adaptation policies. Adaptation policies are meant to change the behavior of the self-adaptive computing system by working on a set of parameters that can belong to either the computing system or the applications, and tuning their influence in accordance with measures and goals retrieved through monitors.

The role of the adaptation manager in the system economy is challenging and important, yet not fundamental [7]. It stands at the center of the higher-level decision loop and exploits the awareness given by the available monitors to elaborate a plan for future behavior. The aim is to tune performance in order to make each monitored process achieve its performance goals. In particular, the adaptation manager is in charge of: determine and constantly updating the available adaptation policies and the

monitored applications; gathering the information coming from the monitored processes; analyzes the performance-related data in order to understand whether to enact a correction policy; decides which application are to be allocated (de-allocated) to (from) adaptation policies and communicate them this decisions. The decision policy that drives the adaptation manager is based on *machine learning* techniques, which were proved powerful tools for managing the increasing complexity of computing systems [51]. The design principles that guided the implementation of the adaptation manager are the following:

- the interface between the adaptation manager and the adaptation policies is extremely simple, allowing fast and low-overhead communication;
- there is no need to model the behavior of the adaptation policies, and for the adaptation manager to know its effects in advance: this capability is provided by the machine learning engine.

The research on this component is still ongoing and, at the time this thesis has been written, a stable version of it is not available. For this reason, the system used for the implementation does not show a adaptation manager, relying only on the lower-level ODA loop implemented within each adaptation policy.

3.1.3 CHANGE over Linux

In order to realize a first prototype of the sketched operating system, named Autonomic Operating System (AcOS), the GNU/Linux OS was chosen as a starting point. The advantages of relying on an already established open-source OS instead of starting writing the whole system from scratch are summarized below:

- 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 diffusion of Linux in many environments (from mobile and embedded devices to servers and supercomputers) offers support for a wide set of architectures where the autonomic OS can natively run.
- 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.
- Linux, through its developers, already addresses the major issues with contemporary operating system. Thus, the CHANGE group can be more focused on the autonomic features, without the need to invest too much time in other issues that would require a lot of attention in a OS developed from scratch.

The next two sections focus more specifically on the work developed for this thesis: the design both of a monitor, able to provide the system information about the contention of locks within different threads, and of a adaptation policy which exploits the information provided by this monitor in order to improve application performance, acting on task mapping within the Linux kernel.

3.2 HRM for Contention Monitoring

The Heart Rate Monitor (HRM) was presented and deeply analyzed in Section 2.2.1, introducing it as a framework for applications performance

monitoring. The approach proposed by HRM and, before, by Application Heartbeats, can have a wider interpretation, since provides the applications a way to communicate that something is happening with a certain frequency, not strictly coupled with the concept of the progress of the job they are performing. It is possible to state that the meaning of an *heartbeat* emission and of the *heart rate* value depend on the interpretation the adaptation policy gives to it. Having this idea in mind, it is possible to interpret the emission of an heartbeat as a communication by a task which notifies that it is stuck waiting for a lock to be released and cannot go on with its job.

This new meaning of an heartbeat inspired the research work presented here and demonstrates the flexibility of the HRM framework. When dealing with HRM for lock contention measuring, an heartbeat is emitted every time a task is not able to access its critical section, since another task is holding the lock. In this scenario, some endpoints of the HRM performance version need a review:

- if when considering performance a greater heart rate means a better application behaviour, a lower heat-rate represents better performance if lock contention is considered;
- the definition of a maximum and minimum heart rate is quite useless if dealing with contention, since the desired heart rate value should be the lowest possible. In the case of performance monitoring, the definition of a minimum heart rate threshold was fundamental to tune the system parameters and, when possible, strictly respected. The maximum heart rate value, on the other hand, was to be set in order to fix an upper bound to the required quality of service, in order to avoid useless computation and, consequently, a temperature increase or an energy waste. However, meeting this constraint was not considered to be crucial.

- a desired heart rate equal for all the applications and not depending on them is 0. In fact, if an heart rate of 0 is reached, it means that no tasks is trying to acquire a lock without being able to do that: thus, there is no contention between threads.

Intuitively, the idea is to write a lock library instrumented with HRM and emitting one heartbeat every time the acquisition of a lock fails. In particular, it makes sense to emit an heartbeat not every time a lock acquisition fails, but if and only if the lock acquisition fails and the task holding the lock is actually executing on another processor.

To demonstrate the validity of the proposed approach a simple lock library was implemented, containing only spin-locks. As described in Section 1.2.4, spin-locks are one of the simplest implementation of locks: they simply wait for the lock variable to change its value and continuously test it, in a *busy waiting* fashion. Spin-locks are still widely used, while their use is usually coupled with other synchronization methods. For these reasons, a library lock containing spin locks is enough to demonstrated that the described methodology is valid, while it would be advisable to extend the instrumented lock library with other synchronization methods to obtain a better description of threads contention.
In order to put into practice the described methodology, each lock is to be associated with a HRM group, allowing the system to retrieve an heart rate for each group (i.e., a number describing the contention over the associated lock). However, the current implementation of HRM does not allow to do this, since, as mentioned before, groups are non-intersecting subsets: a task belongs to only one group at a time (the support for multi-task groups come by default with HRM). In the envisioned approach, each lock must be associated with a group: by maintaining the previous implementation, it would be possible to consider applications contending only for a single lock, which is not a very common case in real applications. Thus, a redesign

of the HRM framework was needed in order to insert this new feature in it (implementation details are discussed in Section 4.1).

Even after these modifications, the HRM monitoring infrastructure maintain some basic design goals:

- instrumented applications performance should suffer as little as possible from the monitoring overhead;
- non-instrumented applications performance should not suffer at all from the presence of monitored applications;
- the information gathered by the monitor should be easily accessible by other interested system components, both in kernel-space and in user-space.

These design principles are met thanks to some wise implementation choices. First of all, the computation of the statistics is decoupled from the emission of the heartbeats: this was done by moving the code in charge of doing the calculation in a routine periodically executed thanks to the high resolution timers available within the Linux kernel. Moreover, the tasks registration to a group is made possible through a clear Application Programming Interface (API) (only partially modified for the introduction of multi-group registration support): in this way the monitoring is activated only on interested application, making the direct overhead on non monitored applications zero.
The gathered information is shared with the interested components in user-space exploiting the great support to shared data structures provided by Linux. This mechanisms proved to be simple and convenient in many cases, while leading to possible performance issues in multi-core and multi-processor systems where a poor design of shared data may generate useless traffic on the buses to maintain cache coherency. The communication of the monitored information is even simpler within the kernel, where a global

groups list is stored and easily accessible. Last, it is worth noting that the heartbeats emission is, also, completely lock-less: locks are needed only for adding and removing tasks to/from a group and for guaranteeing the correctness of the statistics computation.

The statistics that are made available by the monitor are exactly the same exposed by the original HRM and are computed for each group:

- the *global heart rate*, defined as the number of heartbeats emitted by all the member of a given group divided by the total monitoring time. More formally, the global heart rate is:

$$\text{global\_hr}_n(t) = \frac{\sum_{i=0}^{N} \text{cntr}_i(t)}{t} \qquad \left[\frac{heartbeats}{seconds}\right]$$

  where n is the Group IDentifier (GID), t is the time, in seconds, passed since the creation of the group, and the index i goes from 0 to N (the number of tasks in the group).

- the *window heart rate*, calculated on the last W time slots, i.e. on the last $W\times$ *timeslot_duration* seconds. With a mathematical formula:

$$\text{window\_hr}_n(t) = \frac{\sum_{i=0}^{N} \text{cntr}_i(t) - \sum_{i=0}^{N} \text{cntr}_i(t-w)}{w} \qquad \left[\frac{heartbeats}{seconds}\right]$$

  where all the variables have the same meaning as before and w is the effective duration, in seconds, of the time slot.

Next section is devoted to the explanation of the algorithms implemented in order to reduce contention, thus improving application performances.

3.3 Adaptation Policies

In order to exploit the locality of the critical section, data tasks that require the same lock should be moved to the processor that is executing the

task that currently holds the lock. The advantages of implementing such a policy are several. For example, when a lock is released and the thread that blocked on the lock is awakened, it directly uses the data in its local cache rather than doing remote bus requests to fetch the data [75]. This is quite important, given the fact that the current microprocessors are much faster than the memory subsystem and the system bus, and hence have to use the data in their cache very efficiently. Additionally, this translates into a reduction in the number of requests in the bus and hence improvement in the scalability of Symmetric Multi-Processor (SMP) systems. The thread is then able to perform the computation in the critical section faster and to release the lock quicker, since the data is already present in its local cache. All these advantages should speed up applications that have heavy synchronization overhead.

According to these concepts, the adaptation policy should be able to move tasks that contend for the same lock on the same processor. By design, if all threads that share the same lock are moved on the same execution unit, no more heartbeats will be generated within that group, leading to a decreasing global heart rate and null window heart rate. This is due to the fact that if a thread is not able to acquire a lock it is not possible for the lock owner to be in execution, since all the threads are mapped to the same processor. Thus, the adaptation policy that has to be implement should aim at reaching a zero window heart rate for each existing group, remembering that an higher heart rate corresponds to an higher contention on the considered lock.

3.3.1 Lock Contention Data

Before looking at the adaptation policies, it is worth investigating which are and how are organized the data that can be exploited. The implemented system gathers information about the current status of the instrumented

processes periodically. This period can be varied: in the performed experiments this value, after some tuning, was set to 1 second, which provides a good trade-off between the algorithm reactivity and its overhead. Contention data are read from the monitor and stored in specific data structures before starting operating on them, with the specific aim of reducing memory accesses and of optimize storing space. The available data structure are the following:

- `group_array`: an array with a dynamic length which is constantly updated in order to contain the GIDs of the groups active in the system;
- `heartrate_array`: an array of the same length of the `group_array` storing their respective window heart rate;
- `sorted_group_array`: another array of the same length of the previous two, containing the indexes of the first one so that they are sorted by decreasing window heart rate
- `task_array`: an array containing pointers to the `task_struct` describing each instrumented task running in the system;
- `incidence_matrix`: a $n \times m$ matrix, where n is the number of monitored tasks and m is the number of active groups. The cell (i, j) in the matrix is equal to 1 if and only if the group g_j contains the task t_i.

Just to clarify the concepts, Figure 3.2 proposes a visualization of the storing structures presented above.

3.3.2 Implemented Heuristics

Two different adaptation policies where designed for this work of thesis, both based on simple heuristics. The first and the simplest one does not

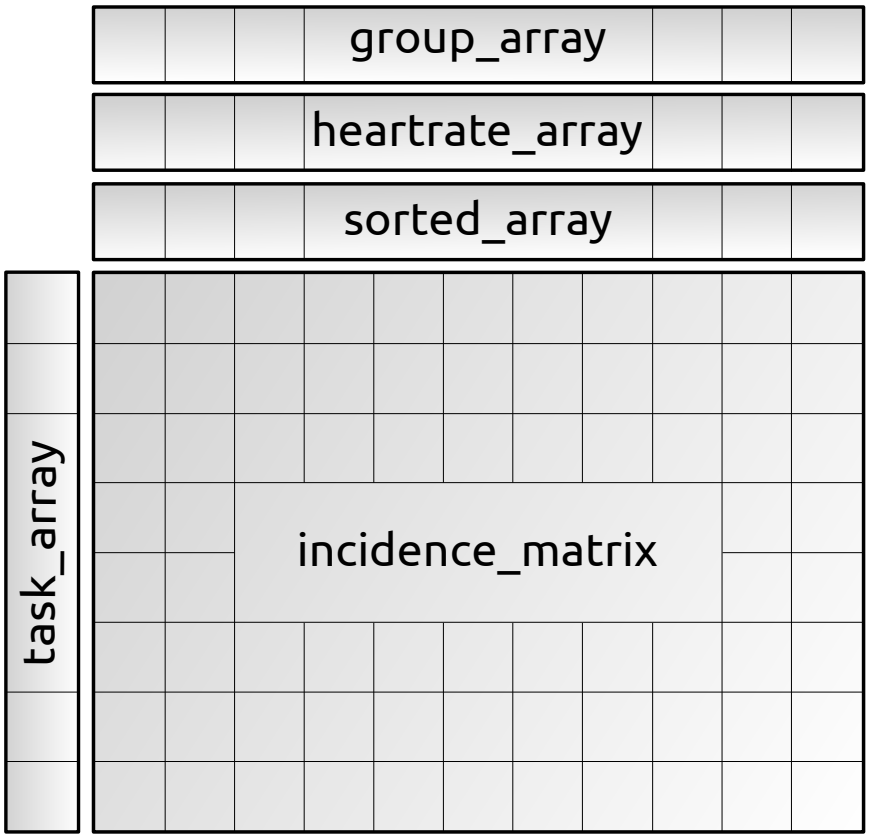

Figure 3.2: Data structures used to store information about the monitored tasks.

use all the described structures: in particular, it does not sort the `group_array` at each iteration, but it simply find the group with the highest window heart rate, neglecting the others. Once the group has been selected, the `incidence_matrix` is parsed in order to find all the tasks belonging to that group. All these tasks are then moved to the same processor. A CPU counter is then increased and the cycle starts again. The pseudo-code for this simple heuristics is shown in Listing 1.

The second heuristics is more complex. At each iteration the `sorted_group_array` is filled in with the indexes of the group_array, sorted by decreasing value of the corresponding window heart rate. Then, for each group, following the order of the sorted array, the tasks belonging to that group are retrieved. All these tasks are again, moved to the same

Algorithm 1 Pseudo-code for the first heuristics.

```
initialize cpu_counter to 0
loop
  update contention data
  find max in the heart_rate_array
  store the index of this element in j
  for all i in task_array do
    if incidence_matrix(i,j) == 1 then
      move task i to the processor cpu_counter % online_cpus
    end if
  end for
  increment cpu_counter
end loop
```

processor. When a task is selected for the first time, its row in the incidence matrix is set to all zeros in order to avoid the policy to move the task again, to accomplish the needs of other groups with a lower heart rate. Again, a CPU counter is used and incremented in order to equally distribute the tasks on all the available processors. The pseudo-code also for this second heuristics has been written and it is reported in reported in Listing 2.

The first approach proved not to provide good performance when dealing with an high number of groups, since the number of cycles required to spread the tasks among all the CPUs was too high. For this reason, after the first experiments, this first implementation was abandoned in favour of the second one which showed better performance, as shown later in Chapter 5.

Execution Example A practical example of this second implementation can help in understanding how it works. Imagine to have 7 tasks, attached to 3 different groups as shown in the incidence matrix in Table 3.1.

When the adaptation policy notices the presence of new groups in the sys-

Algorithm 2 Pseudo-code for the second heuristics.

```
initialize cpu_counter to 0
loop
  update contention data
  sort the heart_rate_array
  store the sorted indexes in the sorted_group_array
  for all i in sorted_group_array do
    for all j in task_array do
      if c == 1 then
        move task with index i to the processor cpu_counter mod
        online_cpus
        for all k in group_array do
          set incidence_matrix(i,j) = 0
        end for
      end if
    end for
    increment cpu_counter
  end for
end loop
```

GID	10	20	30
window hr	/	/	/
sorted index	/	/	/
PID		**CPU cnt**	0
10000	0	0	1
10001	1	0	1
10002	1	1	1
10003	0	1	0
10004	0	1	0
10005	1	0	0
10006	1	0	0

Table 3.1: Adaptation policy example: starting scenario.

tem, all the data structures are filled in, retrieving the window heart rate of each group, and computing the array containing the their sorted indexes. The data structures at this step of execution are shown in Table 3.2.

Now, the heuristics starts doing its job: the group with the highest window heart rate is selected (with GID 20 in this example) and the three tasks belonging to it (with PID 10002, 10003 and 10004, respectively) are moved on the same processor. The rows related to these tasks are filled with zeros and the CPU counter is incremented. Following the window heart rate in a decreasing order, the group with GID 30 is considered: the tasks still belonging to this group according to the incidence matrix are the ones with PID 10000 and 10001. They are moved to the processor with ID 1, the corresponding rows are set to 0 and the CPU counter is newly incremented. Last, the two remaining tasks are moved to CPU 2. The first iteration of the cycle ends since no more groups are to be analyzed: after the selected period the algorithm is executed again, working on updated data.

GID	10	20	30
window hr	10K	30K	20K
sorted index	2	0	1
PID		**CPU cnt**	0
10000	0	0	1
10001	1	0	1
10002	1	1	1
10003	0	1	0
10004	0	1	0
10005	1	0	0
10006	1	0	0

Table 3.2: Adaptation policy example: data structures initialization.

3.4 Summary

This chapter presented the original contribution of this work to the CHANGE project and to the autonomic operating system field. In particular, the attention was focused on the explanation of the theoretical concept behind the developed work. Section 3.1 presented the design principles that inspired the creation of the AcOS operating system and its conceptual structure, in which this thesis is located. The focus was then moved to the methodology that was followed in the design of the lock contention monitoring infrastructure and in the instrumentation of a simple locking library, in Section 3.2. Last, Section 3.3 described the adaptation policy for the tasks performance improvement from an high-level point of view, providing a simple example in order to better understand its behavior. Implementation details of the whole framework, made up of both the monitoring infrastructure and the adaptation policy, are presented in the next Chapter 4.

GID	10	20	30	10	20	30
window hr	10K	30K	20K	10K	30K	20K
sorted index	2	0	1	2	0	1
PID		**CPU cnt**	0		**CPU cnt**	1
10000	0	0	1	0	0	**1**
10001	1	0	1	1	0	**1**
10002	1	**1**	1	*0*	*0*	*0*
10003	0	**1**	0	*0*	*0*	*0*
10004	0	**1**	0	*0*	*0*	*0*
10005	1	0	0	1	0	0
10006	1	0	0	1	0	0

10	20	30	10	20	30
10K	30K	20K	10K	30K	20K
2	0	1	2	0	1
	CPU cnt	2		**CPU cnt**	2
0	*0*	*0*	0	0	0
0	*0*	*0*	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
1	0	0	*0*	*0*	*0*
1	0	0	*0*	*0*	*0*

Table 3.3: Adaptation policy example: heuristics.

Chapter 4

Proposed Implementation

This chapter describes the main technical and implementation details which allowed to put into practice the concepts presented in the previous chapter. This practical part required both to understand and modify already existing code, and also to design and write part of the code from scratch. It was also required to implement code both in user-space and in kernel-space. In particular, the extension of the Heart Rate Monitor (HRM) framework (see Section 4.1) needed a lot of modification in kernel-space, while leaving the user-space library quite unmodified. The lock library, presented in Section 4.2, was completely developed in user-space, while the adaptation policies (Section 4.3) were coded in kernel-space.

As already pointed out, the whole framework is developed within the GNU/Linux operating system, with the 2.6.35.14 kernel version [77]. The 2.6.35 version of the kernel was chosen since it was the latest *longterm* release at the time this work of thesis began (and it is still the latest long term release at the time this thesis is written). In Linux lingo if a release if marked as *longterm*, it means that it will be supported with bug fixes and security patches for a longer time than standard stable releases (in fact, the last final *.14* version number is an incremental number used for updates to the base long term release, registered as 2.6.35).

4.1 HRM Extension

The first step performed to get in touch with the already developed Heart Rate Monitor (HRM) library was writing a patch for the 2.6.35.14 kernel version, since the at the time it was available up to the 2.6.35.13 kernel release. This task was quite straightforward, since the differences in the two releases were not related to files and structures interesting the monitoring library. However, this was a great occasion to become familiar both with the Linux kernel and with the library itself, starting to understand *what* and *where* was to be added code in order to allow one task to be in many groups.
The HRM framework is made up of a kernel-space implementation which does mainly all the job and a user-space library that exposes the functions needed to exploit the features implemented in kernel-space. The main modification that were needed to introduce to possibility for a task to attach to more than one group can be summarized as follow:

- the `task_struct` data structure, which contains all the important information about a task and is located in *include/linux/sched.h*, was modified to store a list of data structures about groups (named `hrm_struct`), instead of only one;
- the main monitoring structure, `hrm_struct` in *include/linux/hrm.h* (the one representing the real monitor: one for each group and for each task) was integrated with some fields allowing to better managing the new features;
- all the related functions, in *kernel/hrm.c* and *fs/proc/base.c*, and their definitions, in *include/linux/hrm.h*, were modified in order to be consistent with the new concepts.

```
1 struct hrm_struct {
```

```
2         int counter_index;
3
4         struct hrm_counter      *counter;
5         struct hrm_stats        *stats;
6         struct hrm_target       *target;
7
8         struct hrm_group        *group;
9
10        unsigned long counter_user_address;
11        unsigned long stats_user_address;
12        unsigned long target_user_address;
13
14        struct task_struct      *task;
15
16        struct list_head link_group;
17        struct list_head link_monitor;
18 };
```

Listing 4.1: The `hrm_struct` data structure in *include/linux/hrm.h*.

The `hrm_struct` in Listing 4.1 is the data structure that allows the monitored information to be made available in the kernel both for periodic statistics computation and for the kernel components interested in accessing it.

It contains information about the group counter, the computed statistics and the group goals (`*counter`, `*stats`, and `*target`, respectively) and the relative memory addresses to allow data to be read from user-space components. `link_group` is a pointer to a global list containing all the groups currently available in the system. All these fields were inherited from the previous implementation of HRM and maintained, even if some are useless in this case, in order to guarantee the compatibility with the performance use of the monitor. The new fields are underlined in Listing 4.1 and their use is explained below:

- `struct task_struct *task`: this field allows to store a pointer to the `task_struct` of the task the `hrm_struct` is coupled with. The previous implementation allowed, by browsing the `*counter`

pointer to retrieve the PID of the task; however, it was absolutely not convenient to access the PID and then retrieve the `task_struct` from this, when it is easily available in kernel and the storing of this pointer does not required a big overhead.

- `struct list_head link_monitor`: is a pointer to a list storing the `hrm_struct` data structures of all the other groups the current task is attached to. It allows to easily go through all the groups a task is part of without retrieving all the existing groups before.

Some helper functions are exported by *include/linux/hrm.h* in order to make the use of HRM capabilities in kernel easier.

```
1 int hrm_add_task_to_group(struct task_struct *task, int gid);
2 int hrm_delete_task_from_group(struct task_struct *task, int gid);
3 int hrm_task_is_enabled(struct task_struct *task);
4 int hrm_task_is_enabled_group(struct task_struct *task, int gid);
5 int hrm_task_is_active(struct task_struct *task, int gid);
```

Listing 4.2: Helper functions exported by *include/linux/hrm.h*.

This functions signatures have been slightly modified in the new HRM version. First of all, the concepts of a task being *enabled* and *active* have a new meaning. A task is said to be simply enabled if it exists at least one group it is attached to; the fact to be enabled can be also restricted to single group: a task is said to per enabled for the group with GID `gid` if it is attached to that group. This is the reason why the function named `hrm_task_is_enabled_group()` was introduced. A task is considered to be active with reference to a group with GID `gid` if its counter is set to active, i.e. when it is within an hotspot.
The `hrm_delete_task_from_group()` function has a new parameter, `int gid`, which represents the GID of the group the task is to be removed from. Obviously this parameter was not needed in the previous implementation, since a task was able to attach only to a single group.

`hrm_add_task_to_group()` maintains the same signature as before, but necessarily its implementation is changed due to the need of keeping all the lists containing the `hrm_struct` updates.
A global variable named `hrm_groups_count` was also introduced, in order to store the number of existing groups, and kept updated every time a groups is created or destroyed. The presence of this variable is needed to make the creation of the data structures used by the adaptation policy more optimized.

The modifications introduced in kernel-space reflects in a few corrections in *libhrm*, the user-space library that allows user space application to use the monitoring infrastructure. These modifications are not that interesting and deserve not to be further investigated here. Simply, the need for the GID to be specified when detaching from a group is made explicit and the related functions writing (in user-space) and reading (in kernel-space) the */procfs* were changed accordingly.

4.2 Lock Library Implementation and Instrumentation

In order to test the validity of the approach explained in the previous chapter, it was necessary to instrument a lock library with the new HRM monitoring infrastructure, allowing a single task to attach to more than one group. The doubt was about writing a lock library completely from scratch or trying to adapt an already existing library. The latter option was discarded because it was not easy to find trustable code to start from and the GNU/Linux lock library (implemented in the *glibc*) proved to be too much optimized and offered too many features to make the instrumentation feasible. For these reasons, it was decided to create a new lock library from scratch, implementing only the functionalities needed to prove the validity and the feasibility of the proposed approach. Thus, the attention was

focused on spin-locks, which represents the simplest type of memory synchronization methods that allows the presented research to work.

The data structure which defines the spin-lock is simple and contains only two fields: a volatile bool named `lock`, and a variable `pid_lock` of type `pid_t` (see Listing 4.3). The former, is the variable which says if there is any task currently holding the lock;the latter field contains the PID of the task, if any, currently holding the lock.

```
1 typedef struct lock_t{
2         bool lock;
3         pid_t pid_lock;
4 };
5 typedef struct lock_t lock;
```

Listing 4.3: Spin-lock type definition.

The functionalities exposed by the library are the basic ones expected for a lock library to provide. In particular, an initialization function and the two functions to acquire (or try to acquire) and release a lock (Listing 4.4). Each function is further analyzed in the remainder of this section.

```
1 void init(lock_t *L);
2 void acquire(lock_t *L, hrm_t *monitor);
3 void release(lock_t *L);
```

Listing 4.4: Function prototypes exposed by the lock library.

The initialization function, as can be seen in Listing 4.5, simply takes the pointer to a `lock` and initialize its fields to meaningful values: in particular, the lock is set not to be hold and the PID of the task holding the lock, which does not exist, is set to 0.

```
1 void init(lock_t *L)
2 {
3         L->lock = false;
4         L->pid_lock = 0;
5 }
```

Listing 4.5: Spin-lock initialization function.

More interesting is the lock acquisition function, fully report in Listing 4.6. The atomic instruction `__sync_lock_test_and_set()` (see Section 1.2.4) is exploited to atomically test the value of the lock variable and to set it to `true` if possible, in order to avoid race conditions to appear. `__sync_lock_test_and_set()` returns the value previously stored in memory: thus, in the case the lock is already hold by another task, the execution enters the while loop. For the reasons already exposed in the previous chapter, it is meaningful for a task to emits heartbeats if and the only if the task currently holding the lock is currently *running*. However this information is not available in user-space: in this context it is possible to know only if a task is *runnable*. A task is said to be runnable either if it currently running or it is on a runqueue waiting to run.

```
1  void acquire(lock_t *L, hrm_t *monitor)
2  {
3          while ( __sync_lock_test_and_set(&L->lock, true) ){
4                  if (syscall(__NR_isrunning, L->pid_lock)) {
5                          heartbeat(monitor);
6                  }
7                  else {
8                          pthread_yield();
9                  }
10         }
11         L->pid_lock = (pid_t) syscall(__NR_gettid);
12 }
```

Listing 4.6: Spin-lock acquisition function.

The information about a task effectively running or not is available only in kernel-space: for this reason a new *syscall* was implemented to expose this information also in user-space. Listing 4.7 shows the code for the new system call: taking as input parameter the PID of the task, it checks if it is

running on one of the available CPUs and returns a value accordingly.

```
1  SYSCALL_DEFINE1(isrunning, pid_t, pid){
2          struct task_struct *task;
3          int i;
4
5          task = find_task_by_vpid(pid);
6          for(i = 0; i < get_present_cpus(); i++) {
7                          if(task_running(cpu_rq(i), task)) {
8                          return 1;
9                  }
10         }
11         return 0;
12 }
```

Listing 4.7: Implementation of the `isrunning()` syscall in *kernel/sched.c*.

Going back to Listing 4.6, if the task holding the lock is running the task executing the code emits an heartbeat meaning that it is stuck waiting for it and continues to do that in a busy waiting fashion, until it is preempted or it is able to acquire the lock. On the other hand, if the PID stored in the lock structure belongs to a task that is not running the current task yields the CPU it is running on, without emitting any heartbeat. Thus, it is useless for it to continue waiting if it wont have the possibility to acquire the desired lock. When the task is able to acquire the lock saves its PID in the provided field in the lock structure, retrieving it thanks to another *syscall*, this time already exported by the kernel. Obviously, the acquire function takes as input parameter also an `hrm_t *monitor`, which represents the monitor on which heartbeats are to be emitted.

Last, the lock release function was implemented (Listing 4.8). It is kept as simple as possible: the lock variable value is set to `false`, without the need to use atomic instruction, since it is not possible for race conditions to arise (the lock is hold by only one task a time and there is no possibility for two different tasks to try to release the same lock at the same time).

```
1 void release(lock_t *L)
2 {
3         __sync_lock_test_and_set(&L->lock, false);
4 }
```

Listing 4.8: Spin-lock release function

For an application to be instrumented with this framework, it has to make use of this lock library and of the *libhrm* too, in order to create the `hrm_t *monitor` to be passed to the lock library.

4.3 Kernel-space Adaptation Policy Implementation

The actual implementation of the adaptation policy is the one that required most of the efforts spent in this thesis work. Mainly not in the design of a suitable heuristics, but in understanding:

- where and when to insert the periodical gathering of the contention data from the monitor infrastructure: the choice of the instant in which to perform this task is not trivial at all and must be carefully investigated;
- how to actually move tasks from one runqueue to another, in a *legal* way.

These are the two main issues this paragraph addresses and to which tries to find a solution.

4.3.1 *When* and *Where*

Choosing the proper time in which to let the adaptation policy act proved not to be an easy task. Different roads were pursued before finding the one which seems to be the best available one:

- *kernel tick*. The time inside the kernel is beaten by a tick, which is emitted with a period of 1 millisecond. A timer interrupt handler exists and is invoked every time one tick is charged to the current process. This component calls a `update_process_times()` function (located in *kernel/timer.c*), which is in charge of performing all the periodical operations related to the kernel tick, in particular: run local kernel timers, run POSIX CPU timers, and propagate the tick to the scheduler. The first idea was to add a function performing the work of mapping threads on core according to contention information in this function. However, in this context was not possible, due to the high number of locks hold, to efficiently modify the affinity mask of the task. Thus, the only possible actuation was to directly move task from one runqueue to another. Unfortunately this approach, as better explained later, has a serious drawback and was discarded.

- *high resolution timers*. High resolution timers, also known with the name of *hrtimer*, provide an infrastructure for the implementation of timers with a resolution up to a 1 nanosecond. Thanks to them it is possible to schedule a function to be executed periodically inside the kernel. This was exactly what it was needed. However, *hrtimers* have the not really known drawback of executing always on the same CPU: in particular, the one on which they were initialized. This fact, coupled with the limitation of moving tasks from one runqueue to another, implied the need for creating a timer for each CPU available in the system and to synchronize them. This approach, however, proved to have an high overhead on the system and was, then, discarded.

- *kernel threads*. This last option was, finally, the chosen one. Kernel threads are, as the name suggests, an implementation of threads in kernel-space. The advantages of this approach is that the thread func-

tion is executed outside the context of any other task, thus allowing to operate in an environment where locks can be easily managed.

Once that the suitable technique to periodically manage the available data about contention has been chosen, it is possible to explain how it was implemented. First of all a new file, *kernel/contention.c*, was created and added to the compilation tool-chain. The initialization of the kernel thread is done through a `module_init()` call, which invoke the `kthread_contention_init()` function, showed in Listing 4.9. Here the kernel thread is initialized to call the `__manage_contention()` function, where an infinite loop executes a sleep of `CONTENTION_CHECK_PERIOD` milliseconds and then call the main function for managing the contention: `manage_contention()`.

```
1  static int __init kthread_contention_init(void)
2  {
3          struct task_struct *ts;
4
5          ts = kthread_run(__manage_contention, NULL, "Contention thread");
6          return 0;
7  }
8
9  static void __manage_contention(void)
10 {
11         for(;;)
12         {
13                 msleep(CONTENTION_CHECK_PERIOD);
14                 manage_contention();
15         }
16 }
```

Listing 4.9: The kernel thread initialization function.

The function that actually implement the heuristics policy described in the previous chapter can be found in *kernel/sched.c* and is fully reported in Listing 4.10. This function exactly follows the pseudo-code showed in Listing 2. Only some aspects are to be underlined:

- the *initAndPopulateGroupArrays()* is in charge of filling in all the data structures with the data gathered from the HRM monitor at each cycle;
- the lock on the HRM group list is acquired at the beginning of the function in order to read the group list without risks for a race condition, and released at the end;
- `cpu_count` is an integer global variable initialized to 0;
- *sched_setaffinity()* is used to actually map tasks on cores, as will be explained in a few paragraphs.

```
1  int manage_contention(void) {
2          struct hrm_group *group;
3          struct hrm_struct *hrm;
4          struct hrm_stats *stats;
5          int *grp_incid_matrix, *grp_array, *ordered_grp_array,
               *heartrate_array;
6          struct task_struct **task_array, *task;
7          struct rq *curr_rq;
8          unsigned long groups_lock_flags;
9          int index, curr_cpu, groups_count_loc, i, j, tmp;
10         cpumask_var_t new_mask;
11
12         read_lock_irqsave(&hrm_groups_lock, groups_lock_flags);
13
14         if (hrm_groups_count > 0) {
15           /*
16             ... data structure allocation ...
17           */
18           initAndPopulateGroupArrays(grp_incid_matrix, grp_array,
                 task_array, heartrate_array);
19           groups_count_loc = hrm_groups_count;
20           read_unlock_irqrestore(&hrm_groups_lock, groups_lock_flags);
21           /*
22             ... sorting groups according to their window heart rate ...
23           */
24           for (i = 0; i < groups_count_loc ; i++) {
```

```
25              if (heartrate_array[ordered_grp_array[i]] > 0) {
26                    for (j = 0 ; j < MAX_TASKS; j++) {
27                      if (grp_incid_matrix[j * groups_count_loc +
                            ordered_grp_array[i]] == 1) {
28                            cpumask_clear(&new_mask);
29                            cpumask_set_cpu(cpu_count % num_present_cpus(),
                                  &new_mask);
30                            sched_setaffinity(task_array[j]->pid, new_mask);
31                            memset(&grp_incid_matrix[j * groups_count_loc],
                                  0, sizeof(int) * groups_count_loc);
32                            }
33                           }
34                    cpu_count++;
35                  } else {
36                    break;
37          }
38            }
39            /*
40              ... cleaning up the allocated data structures ...
41            */
42          } else {
43            read_unlock_irqrestore(&hrm_groups_lock, groups_lock_flags);
44          }
45          return 0;
46  }
```

Listing 4.10: Heuristics implementation in *kernel/sched.c*.

4.3.2 *How*

Two are the considered ways to map the tasks on the cores: both are briefly presented here.

- Inspired by the work done by the load balancer, an idea is to directly move a certain task from a runqueue to another. The role of the load balancer within the process scheduler is to find the busiest runqueue in the system and move a task from that runqueue to the one the load balancer code is executing. To do that, this component exploits

a function named `pull_task()`, which takes as input parameters the source and the destination runqueues, the destination CPU and the task to be moved. Due to the context in which this function is executed, a major limitation arises: the destination runqueue must be the local runqueue. This means that the task can be moved only on the runqueue of the CPU on which the core is running. This drawback was enough for the use of this function to be avoided.

- The second approach is indirect, in the sense that consists in modifying the affinity mask of the task that has to be moved. The definition of affinity mask has been already given in Section 2.2.2 when talking about Sefos. However, simply setting the affinity mask of the task is not enough, since in this way the task has to wait for the load balancer to be invoked before being actually moved. In order to trigger this mechanism, a specific function, as the one called by `sched_setaffinity()` system call is to be invoked, or the `sched_setaffinity()` itself. However this is possible if and only if this function is executed outside the context of any other task: as the one offered by the kernel thread.

4.4 Summary

The implementation details that allowed to write a completely working system were presented in this chapter. In particular, the modification introduced to the HRM framework in order to allow it to support multi-group and then to be used as a monitor for lock contention were described in Section 4.1. The motivations for the implementation of a simple spin-lock libraries and some interesting details about the exposed functions are summarized in Section 4.2. Last Section 4.3 contains the solutions that have been found in order to solve the two main issues related to the adaptation

policies implementation: when and where add the code that synthesizes the monitoring information and computes the decision and how actually implement the mapping of tasks on the available cores. After having seen how the proposed contention monitor and the adaptation policy have been designed and implemented (in Chapter 3 and Chapter 4, respectively), some preliminary results that validate the effectiveness of the proposed approach are presented in the following Chapter 5.

Chapter 5

Experimental Results

In this chapter the experiments performed and the gathered results are shown and explained. In particular, in Section 5.1 the testing environment on which the experiments were performed are presented. Section 5.2 describes some experiments aiming at proving the validity of the proposed approach through synthetic micro-benchmarks, while Section 5.3 presents some results related to the applications instrumentation overhead. Experiments which explains how the adaptation policy actually works are described in Section 5.4. Last, data related to the real performance improvements obtained in term of reduction of process execution time, are introduced in Section 5.5, analyzing the behaviour of both micro-benchmarks and real world applications.

5.1 Experimental Environment

The Linux kernel used for the implementation of the kernel part of the monitoring infrastructure and for the adaptation policy is the 2.6.35 released, since marked as *longterm*. The development started from the 2.6.35.13 minor release and a patch was written in order to port the HRM monitoring infrastructure to the following 2.6.35.14 release. The modified Linux ver-

sion was used as the kernel of GNU/Linux operating system, in is Debian 6.0 *squeeze* distribution.

The experiments described in the following sections were done on a $x86-64$ machine featuring a quad-core Intel Core $i7-870$ microprocessor running at 2.93 GHz with 8 MB of shared Last Level Cache (LLC), and 4 GB of $DDR3-1066$; this microprocessor supports Simultaneous Multi-Threading (SMT) through Intel Hyper-Treading (HT) technology and offers the Intel Turbo Boost Technology, but these advanced features were disabled for all experiments.

5.2 Preliminary Experiments

In order to validate the approach described in the previous chapters, some specific tests were performed. In particular it would be interesting to analyze how the monitored heart rate changes according to the number of threads that shares a single lock and to the length of the critical section (quantified as the number of instructions executed between the lock acquisition and its release). These experiments were run writing an application instrumented with the developed locking library, but without enabling the adaptation policy in the kernel.

The first aspect to be analyzed is the relation between heart rate and the number of thread effectively contending for a single lock. In the following plot, it is shown the global heart rate when the same simple application is run with a number of threads varying from 2 to 8. The function executed by the threads simply increments a counter after having acquired the lock, and releases it immediately. The number of times the counter is incremented each time is used to vary the length of the critical section. As an example the case in which the counter is incremented 10000 times is presented.

As expected, the global heart rate is proportional to the number of threads

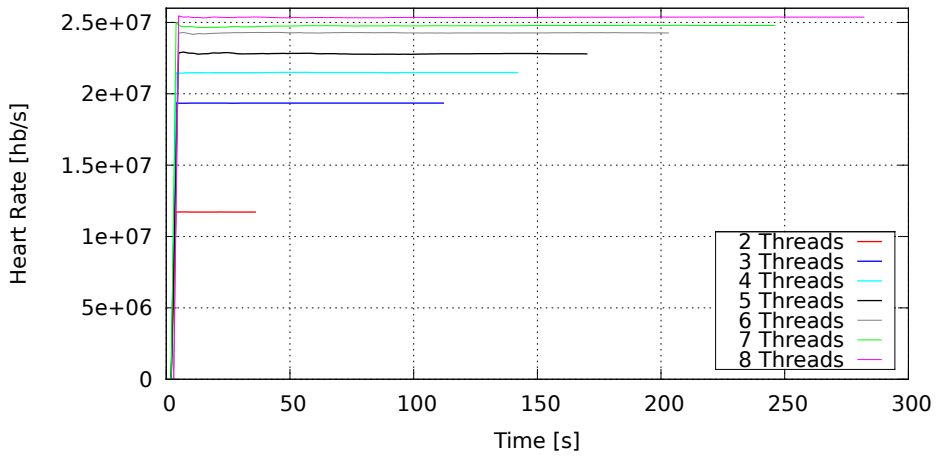

Figure 5.1: Heart rate behavior varying the number of threads (10000 increments in the critical section).

contending for the lock, independently from the length of the critical section. The lowest heart rate is obtained with only two threads sharing the same lock; the value of the heart rate then increases when the number of threads gets bigger. This behaviour is better explained by Figure 5.2, in which the final value of the global heart rate, representing the average heart rate of the application during the whole execution, is plotted.

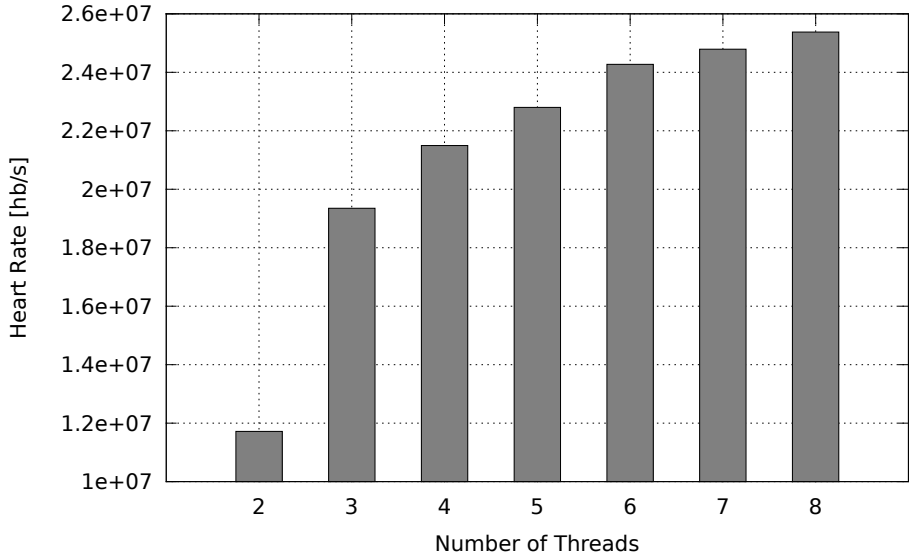

Figure 5.2: Average heart rate behavior varying the number of threads (10000 instructions in the critical section).

The average heart rate increases quite linearly when the number of threads goes from 2 to 4, while the steepness of the fitting curve decreases when the number of threads grows again. This is due to the fact that the the machine on which the experiments are performed has 4 physical cores, thus only for threads can actually run in parallel (remember the Intel Hyper-Treading technology is disabled).

In order to test the behavior of the system according to the length, expressed in number of instructions, of the critical section the same application was used, varying the times the counter is incremented each time the lock is acquired. According to the results shown in the following plots (Figures 5.3, zoomed in Figures 5.4, and Figures 5.5), as expected, the heart rate increases as the length of the critical section increases. However, looking at the value the heart rate assumes when the number of instructions in the critical section is in the order of magnitude of 1, 100, and 10000, it is possible to say that this factor influences the behavior of the heart rate to a lesser extent.

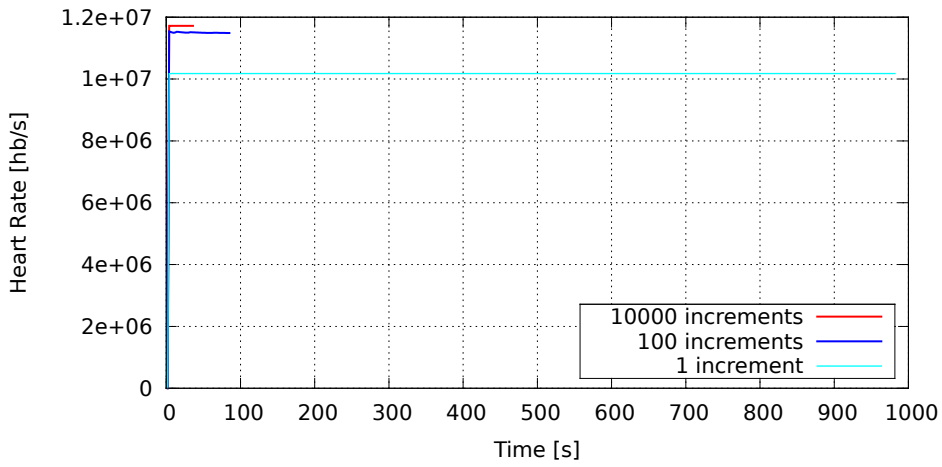

Figure 5.3: Heart rate behavior with 2 threads and 1 a single increment in the critical section.

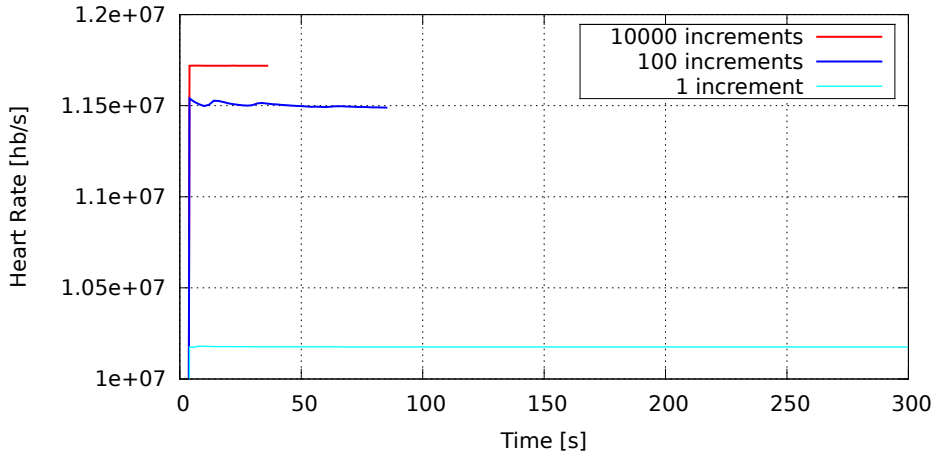

Figure 5.4: Heart rate behavior with 2 threads and 1 a single increment in the critical section – Zoom in.

It is possible to see that the behavior of the heart rate is quite heterogeneous when varying the number of instruction in the critical sections from 1, to 100 to 10000. However, it is clear that the higher the number of threads

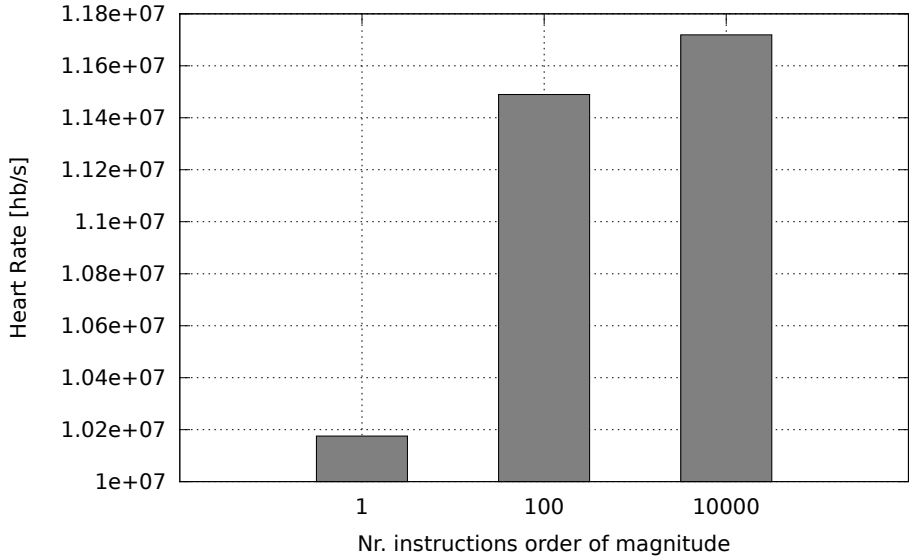

Figure 5.5: Average heart rate behavior with 2 threads (1 increment in the critical section).

sharing a lock is, the higher the measured heart rate of the group related to that lock will be.

Looking at the plots, it is also possible to note how the execution time varies as expected. In particular, in Figure 5.1 the executed time is directly proportional to the number of threads sharing the lock: if a lower number of threads shares the lock, the synchronization overhead is lower, thus the overall execution time. Similarly, from Figure 5.3 it is possible to see that the execution time is lower if the number of increments executed in critical section is higher. This is because the number of synchronization instruction is reduced, and the threshold value of the counter is reached in less time.

5.3 Instrumentation Overhead

One of the first experiments performed to evaluate the modification to the HRM framework and the newly developed locking library was the evaluation of their overhead on real word applications. The benchmark suite chosen to be instrumented is the Stanford Parallel Applications for

Shared Memory (SPLASH) suite [78], in its second review [79], applying the patch provided by the University of Delaware [80] in order to allow the compilation on modern operating systems. The SPLASH-2 suite of parallel applications aims at facilitating the study of centralized and distributed shared-address-space multiprocessors. The suite provide several benchmarks classified by their computational load balance, communication to computation ratio and traffic needs, important working set sizes, and issues related to spatial locality. One of the main targets is to assist people who will use the programs in architectural evaluations to prune the space of application and machine parameters in an informed and meaningful way.

This benchmark suite was chosen since it offer multi-threaded applications sharing a lot of data and makes use, among the others, of spin-locks as synchronization methods. The instrumentation of all the applications proved not to be that easy, and sometimes infeasible. Thus, it was decided to focus the attention on the applications that seemed to be the best suitable one for performing test: *raytrace*.
This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid is used to represent the scene, and early ray termination and antialiasing are implemented. A ray is traced through each pixel in the image plane, and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The major data structures represent rays, ray trees, the hierarchical uni- form grid, task queues, and the primitives that describe the scene. The data access patterns are highly unpredictable in this application.

Many tests were run on the application that were correctly instrumented (by substituting the old synchronization mechanisms with the new spin

lock library) and an overhead spanning the range 2% to 11%, with an average under the 5%. The variation of the overhead is due to the different number of threads spawn by the application, the input file provided to it or the level of anti-aliasing requested by the user. Obviously, this percentage gives only an idea of the overhead, since it depends on the number of lock operations by the specific application. The obtained results are quite positive, since the overhead introduced by the instrumentation is acceptable. Moreover, the time lost with the overhead is, in many cases, widely regained by the advantages introduced by the adaptation policy that exploits this new information.

5.4 Scheduler Adaptivity

The following experiments are aimed at proving and explaining how the adaptation policy works. The working scenario is the following one: 3 different threads (t_1, t_2, and t_3) are running in the system and two locks are contended, lock l_A and lock l_B. t_1 contends only for l_A, t_3 contends only for l_B, while t_2 contends first for l_A, then for l_B. The experiment in two different cases: once with a version of the kernel with the adaptation policy switched off, once with the adaptation policy enabled.

During the execution two HRM groups will be created, since two are the locks on which the three threads contend. t_1 attaches only to group l_A, t_3 only to group l_B, while t_2 to both the groups. Figure 5.6 shows the behavior of the global heart rate (solid line) and the window heart rate (dashed line) of the two groups when the adaptation policy is switched off. The three threads are mapped by the CFS on three different cores (as shown in Figure 5.7) and remain of the same cores during all the execution time. At the beginning the l_A group has a positive window rate, since thread t_1 and t_2 are contending for it. The global heart rate is high until t_2, after about

10 seconds of execution, starts contending for l_B: at this time the window heart rate of the l_A group falls down to zero and its global heart rate starts to decrease. In a similar way,when t_2 starts contending for the other lock the window and global heart rate of the second group show a similar behavior. The window heart rate reaches its maximum value suddenly until the end of the execution, and the global heart rate increases exponentially.

By enabling the adaptation policy in the kernel, the plot that is obtained is reported in Figure 5.8. Figure 5.9 shows how the threads are mapped on the available cores during all the execution time. Again, the threads are mapped by the CFS on three different cores. When t_1 and t_2 start contending for l_A, thus generating a positive heart rate, the adaptation policy moves both the threads to the same core: in this way the window heart rate suddenly falls to 0 and the global heart rate starts decreasing. When t_2 starts contending for l_B a positive heart rate is emitted by the second group: both the global and the window heart rate were zero until that time. The adaptation policy moves the second thread to the same core of the third thread, thus reducing the emitted heart rate: the window heart rate falls to zero, while the global heart rate decreases.

5.5 Performance Improvements

A simple micro-benchmark was implemented in order to show the improvements in term of the reduction of the execution time if the adaptation policy is enabled within the kernel. The micro-benchmark simply consists in a process spawning a variable number of threads. These threads try to increment a counter that is shared among them, thus contending for the lock it is protected with.

Table 5.1 shows the resulting execution times, comparing the case in which the adaptation policy is enabled within the kernel and the case in

which it is not. The data in the *Free execution* column refers to the case in witch the kernel is not augmented with the adaptation policy; data in the *Constrained Execution* column are gathered with the adaptation policy enabled. Each test was repeated 5 times, the reported data are the average ones.

		Free Execution		**Constrained Exec.**		
Threads	**Increm.**	*Avg. Exec. Time* [s]	*Std. Dev.*	*Avg. Exec. Time* [s]	*Std. Dev.*	**Speed-up**
2	100M	23.279	0.032	9.935	0.319	2.343×
2	1G	233.070	1.091	103.395	2.528	2.254×
4	100M	24.346	0.27	20.227	0.752	1.204×
4	1G	242.883	0.92	207.553	2.687	1.170×
8	100M	38.405	1.312	19.868	0.932	1.933×
8	1G	389.087	25.452	415,562	6.556	0.936×

Table 5.1: Execution times improvements on a simple micro-benchmark.

The results showed in the table testifies the validity of the pursued approach and the correctness of the developed implementation. Notifiable improvements are obtained: in particular, the reduction in the applications execution times is higher if the number of threads contending the lock is smaller. Moreover, when the number of threads becomes high with reference to the number of available execution units, other factors start to have more influence on the execution (e.g., context switches). The only case in which a negative speed-up is obtained (i.e., a value lower than 1) is the last one. However, the adaptation policy proves to provide a higher stability to the execution since the obtained standard deviation (6.556 s) is way lower than the free execution case (25.452 s).

In order to test the performance improvement not on a simple micro-benchmark, but on a real application, the already presented *raytrace* application from the SPLASH-2 benchmark suite was considered. The test were performed running a varying number of instances of the application, always with 4 threads each and all with the same parameters related to the input file and the anti-aliasing. Again, the experiments were repeated 5 times: average data and statistics are listed in Table 5.2.

Processes	**Statistics**	**Free Exec.**	**Constrained Exec.**	**Speed-up**
4	*Avg. Exec. Time* [s]	35.08	25.259	1.389×
	Std. Deviation [s]	0.736	1.528	
3	*Avg. Exec. Time* [s]	26.29	25.133	1.046×
	Std. Deviation [s]	4.822	1.634	
2	*Avg. Exec. Time* [s]	17.641	25.042	0.704×
	Std. Deviation [s]	3.849	1.907	

Table 5.2: Execution times improvements on the *raytrace* benchmark.

In the first case, in which 4 instances of *raytrace* are run, all the cores are busy and there is high contention among the different threads: a real speed-up of 1.389× is obtained. This is the best condition in which to obtain improvements: in fact, by reducing the number of processes executing in parallel, the speed-up decreases. This is due to the fact that, in the case cores are not so busy, spawning the different threads of the same application to different cores (thus, not having advantages from sharing the same processor) can lead to better performance (with a negative speed-up of 0.704× in the case of only 2 processes running): in this case the bottleneck is represented by the available executing resources and not by data contention. In conclusion, by looking at the trend, the speed-up is likely to grow in the case the number of threads or processes increases.
It is worth also noting how the execution time in the case of constrained

execution (i.e., when the adaptation policy is enable in the kernel) is quite constant, since all the 4 threads of a process are always moved to one processor, independently of the number of executing processes. On the other hand, the execution time in the case of free execution is clearly decreasing.

5.6 Summary

This chapter presented the experimental results obtained putting into practice the approach and the implementation details described in the previous chapters. First, the environment on which the experiments were performed is described, both the architecture the computing system is equipped with and the operating system (Section 5.1). Different types of experiments were designed in order to validate the proposed approach (Section 5.2), show the overhead in terms of execution time introduced by the instrumentation (Section 5.3), and explain how the adaptation policy actually works (Section 5.4) and which are the performance improvements, both on simple micro-benchmarks and on a real word application (Section 5.5).

Chapter 6, which concludes this dissertation, wraps up the work developed, trying to highlights its major contributions and some criticalities. Moreover, some possible improvements of the approach are proposed and left as future works.

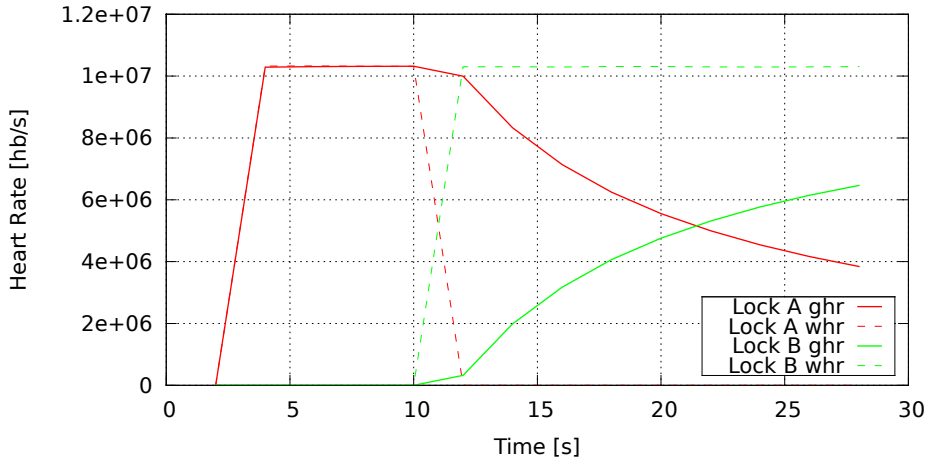

Figure 5.6: Hear-rate for the described execution scenario, when the adaptation policy is switched off.

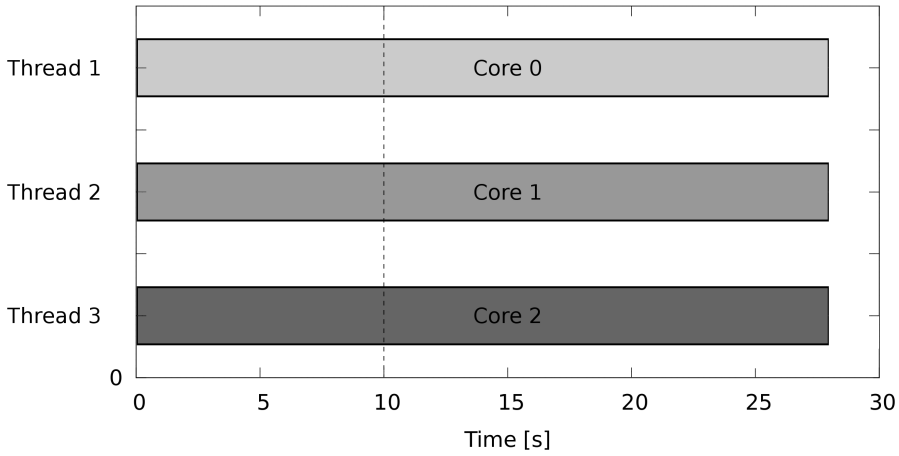

Figure 5.7: Threads mapping on cores, while executing the described scenario, when the adaptation policy is switched off.

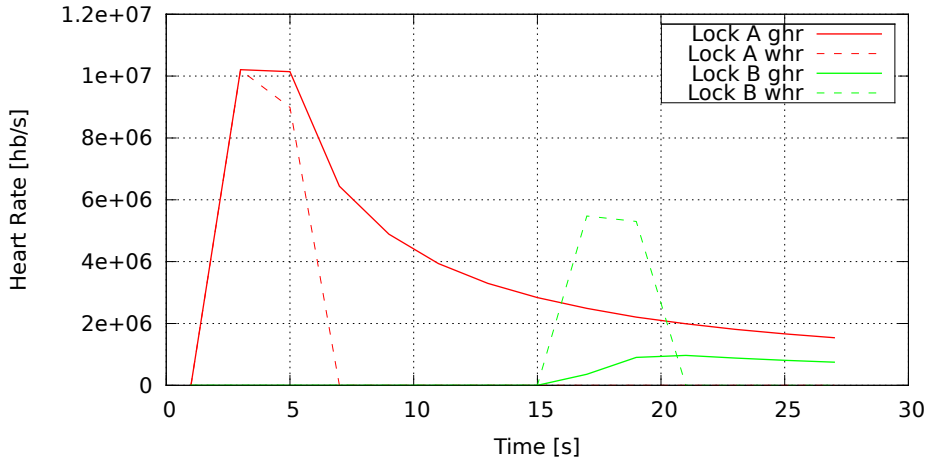

Figure 5.8: Hear-rate for the described execution scenario, when the adaptation policy is switched on.

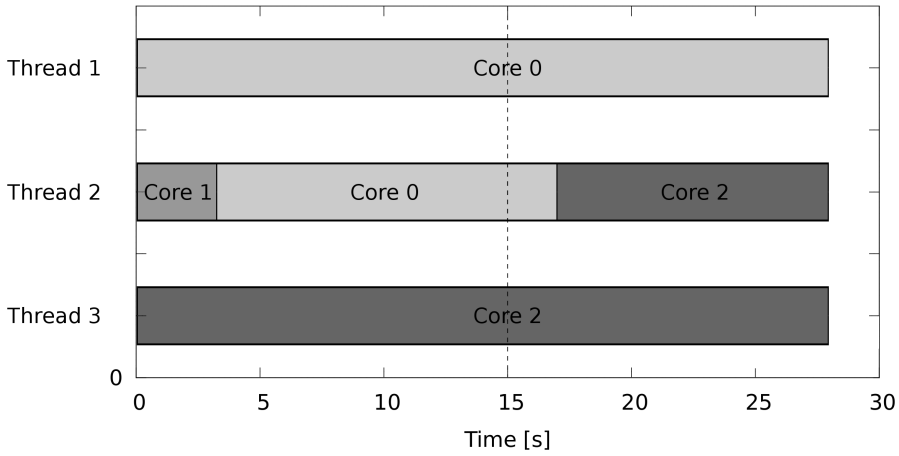

Figure 5.9: Threads mapping on cores, while executing the described scenario, when the adaptation policy is switched on.

Chapter 6

Conclusions

This chapter concludes the description of the developed work. After having analyzed the context of autonomic operating systems and having investigated the state of the art on monitoring and self-aware scheduling, the original contributions of the work were presented and described in the details. These contributions are briefly summarized here.

- Starting from the existing HRM framework, designed for performance monitoring, it was extended to allow a task to attach to more than one group. In this way, it was possible to use this framework as a basis for contention monitoring too, showing the flexibility of the concept of *heartbeat* and *heart rate*.

- A simple locking library was implemented, containing only the functionality necessary to the proposed approach to be validated. This user-space library was then instrumented with the new HRM framework, to enable applications written using it to share information about their shared data contention.

- Two different adaptation policies, based on simple heuristics were designed. These policies decide how to move tasks on the available cores, integrating the work done by the Linux process scheduler. Based

on the fact that tasks running on the same core experience lower communication overhead when they share data, the policy moves tasks with an high heart rate value on the same core, in order to improve their performance.

- The adaptation policies were actually implemented within the Linux kernel, focusing the attention on avoiding the criticalities related to the moment in which the policy acts and how it acts.

The work presented in this document introduced the enabling technology for monitoring contention of data shared among threads and showed how the new information can be exploited in order to insert autonomic capabilities within an operating system. This autonomic capability is represented by the ability of moving tasks on the available cores, according to the information about contention provided by the monitor. The advantage of this autonomic capability can be noticed at the user level in the decreasing of the execution time of threads heavily sharing data with other threads.

6.1 Future Works

The good results obtained and described in Chapter 5 prove the validity of the implement approach and stimulate the research on this field to go on. There are some aspects of the implementation that represent a limitation or that can be further investigated: these are left as future works. In particular:

- at the moment, the implemented lock library allows the applications to use only spin-locks. If an application needs to use more complex synchronization systems has to exploit other libraries, obviously not able to be monitored. Thus, in order to have more precise information about data contention, the instrumented library should be extended to include other synchronization methods, such as condition variables, barriers, ...

- the adaptation policies designed are based on really trivial heuristics. It would be interesting to explore other possibilities for the monitoring data to be exploited: control theory is only on example of the techniques that the policies can rely on.

- in the current implementation, even if both the data related to the global and the window heart rate of the application are exposed by the monitor, only the window heart rate is actually used in the calculation of the heuristics. The global heart rate could be exploited in order to have an idea of the past behavior of the threads contention and to implement a more accurate policy.

- it would be interesting to dynamically vary the period at which the policy is executed, may relating it to the heart rate of the tasks running on the system. At the moment the period is fix and it is necessary to recompile the whole kernel to change it.

Bibliography

[1] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

[2] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology, Oct 2001. [Online] Available: `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`.

[3] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[4] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142, New York, NY, USA, 2010. ACM.

[5] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. *SIGPLAN Not.*, 44:121–132, March 2009.

[6] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 117–, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Marco Triverio. Self-adaptability via heartbeats and central multi-application coordination. Master's thesis, Politecnico di Milano, 2010.

[8] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[9] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[10] Marco Domenico Santambrogio. A scheduling problem with conditional jobs solved by cutting planes and integer linear programming, 2007. Minor PhD Thesis.

[11] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-driven processor allocation. In *OSDI'00*, pages 59–73, 2000.

[12] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, October 1989.

[13] V.C. Hamacher, Z.G. Vranesic, and S.G. Zaky. *Computer organization*. McGraw-Hill series in computer science. McGraw-Hill, 2002.

[14] Intel. Intel 64 bit and ia-32 architectures software developer's manual.

[15] J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 2003.

[16] Various Authors. Ieee std 1003.1, 2004 edition.

[17] Mark Mitchell and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[18] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[19] Henry Hoffmann, Martina Maggio, Marco Domenico Santambrogio, Alberto Leva, and Anant Agarwal. Seec: A framework for self-aware computing. Technical report, Massachussets Institute of Technology, Oct 2010.

[20] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[21] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[22] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Adv. Eng. Inform.*, 19:181–187, July 2005.

[23] Filippo Sironi. Design and implementation of an hot-swap mechanism for adaptive systems. Master's thesis, Politecnico di Milano, 2010.

[24] Anant Agarwal and Markus Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 750–753, New York, NY, USA, 2007. ACM.

[25] K. Modzelewski, J. Miller, A. Belay, N. Beckmann, C. Gruenwald III, D. Wentzlaff, L. Youseff, and A. Agarwal. A Unified Operating System for Clouds and Manycore: fos. 2010.

[26] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 3–14, New York, NY, USA, 2010. ACM.

[27] Silas B. Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 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.

[28] Various Authors. The mit angstrom project: Universal technologies for exascale computing (project home page), Mar 2011. [Online] Available: `http://projects.csail.mit.edu/angstrom/index.html`.

[29] Andrew Baumann, Paul Barham, Pierre E. Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems.

In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[30] A. Schupbach, Peter Simon, Andrew Baumann, and Timothy Roscoe. Embracing diversity in the Barrelfish manycore operating system. 2008.

[31] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[32] Pascal Brisset, Thom Frühwirth, Carmen Gervet, Pierre Lim, Micha Meier, Thierry Le Provost, Joachim Schimpf, and Mark Wallace. Eclipse 3.5 - ecrc common logic programming system - extensions user manual, 1995.

[33] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[34] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[35] Marc Auslander, Dilma Dasilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42 overview. Technical report, IBM T. J. Watson Research Center, Aug 2002. White paper. [Online]. Available: `http://www.research.ibm.com/K42/white-papers/Overview.pdf`.

[36] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with k42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427 –440, 2005.

[37] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 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, pages 87–100, Berkeley, CA, USA, 1999. USENIX Association.

[38] Marc Ausl, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewsk, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. abstract. Technical report, International Business Machines (IBM), Jul 2003. Technical Report RC22863, [Online] Available: `http://domino.research.ibm.com/comm/research.nsf/pages/r.os.innovation.html/$FILE/scalability-techrep.pdf`.

[39] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

[40] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[41] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 3–8, New York, NY, USA, 2002. ACM.

[42] Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da, Silva Orran, Krieger Marc, Auslander David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul Mckenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. Technical report, IBM T. J. Watson Research Center, 2003. [Online]. Available: `http://www.research.ibm.com/K42/white-papers/MemoryMgmt.pdf`.

[43] E. Rotem, J. Hermerding, A. Cohen, and H. Cain. Temperature measurement in the intel core duo processor, 2007. Technical Report.

[44] Russ Joseph, David Brooks, and Margaret Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs. Technical Report.

[45] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 135–140, New York, NY, USA, 2001. ACM.

[46] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[47] Xizhou Feng, Rong Ge, and K.W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 34, april 2005.

[48] P. Bellasi, W. Fornaciari, and D. Siorpaes. Predictive models for multimedia applications power consumption based on use-case and os level analysis. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1446 –1451, april 2009.

[49] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Memory management in k42. Technical report, IBM T. J. Watson Research Center. White paper. [Online]. Available: `http://www.research.ibm.com/K42/white-papers/MemoryMgmt.pdf`.

[50] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. 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, pages 79–88, New York, NY, USA, 2010. ACM.

[51] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization. In *Proceedings of the seventh International Conference on Autonomic Computing*, pages 215–224, 2010.

[52] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A Framework for Self-aware Management of Multicore Resources. Technical Report MIT-CSAIL-TR-2011-016, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 2011.

[53] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16$^{\text{th}}$ International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–212, 2011.

[54] Davide B. Bartolini. Adaptive process scheduling through applications performance monitoring. Master's thesis, UIC - University of Illinois at Chicago, 2011.

[55] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[56] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.

[57] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *in EuroSys*, 2007.

[58] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28:54–66, May 2008.

[59] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[60] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53:49–57, February 2010.

[61] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[62] Inchoon Yeo and Eun Jung Kim. Temperature-aware scheduler based on thermal behavior grouping in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 946–951, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[63] Min Bao, Alexandru Andrei, Petru Eles, and Zebo Peng. On-line thermal aware dynamic voltage scaling for energy optimization with frequency/temperature dependency consideration. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 490–495, New York, NY, USA, 2009. ACM.

[64] P. Bailis, V.J. Reddi, S. Gandhi, D. Brooks, and M. Seltzer. Dimetrodon: Processor-level preventive thermal management via idle cycle injection. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 89 –94, june 2011.

[65] Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. 2005.

[66] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition, 2009.

[67] M. Tim Jones. Inside the linux 2.6 completely fair scheduler, 2009. [Online] Available: `http://public.dhe.ibm.com/software/dw/linux/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf`.

[68] Avinesh Kumar. Multiprocessing with the completely fair scheduler, 2008. [Online] Available: `http://download.boulder.ibm.com/ibmdl/pub/software/dw/linux/l-cfs/l-cfs-pdf.pdf`.

[69] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Decision making in autonomic computing systems: comparison of approaches and techniques. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 201–204, New York, NY, USA, 2011. ACM.

[70] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Scheduling in k42. Technical report. White paper. [Online]. Available: `http://www.research.ibm.com/K42/white-papers/Scheduling.pdf`.

[71] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Performance-driven processor allocation. *IEEE Trans. Parallel Distrib. Syst.*, 16:599–611, July 2005.

[72] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of smart: a scheduler for multimedia applications. *SIGOPS Oper. Syst. Rev.*, 31:184–197, October 1997.

[73] Jason Nieh and Monica S. Lam. A smart scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21:117–163, May 2003.

[74] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[75] Srinivas Sridharan, Brett Keck, Richard Murphy, Surendar Ch, and Peter Kogge. Thread migration to improve synchronization performance. In *In Workshop on Operating System Interference in High Performance Applications*, 2006.

[76] HP. Perfmon project. `http://www.hpl.hp.com/research/linux/perfmon/`.

[77] Linus Torvalds. Linux kernel archives, Mar 2011. [Online] Available: `http://http://www.kernel.org/`.

[78] Jaswinder P Singh, Wolf Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. Technical report, Stanford, CA, USA, 1991.

[79] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[80] Ioannis E. Venetis. The modified splash-2 home page. `http://www.capsl.udel.edu/splash/`.

November 28, 2011

Document typeset with LaTeX