POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

Operating System Support for Adaptive Performance and Thermal Management

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Dott. Ing. Filippo SIRONI

Tesi di Laurea di:

Riccardo Cattaneo

Matricola n. 755279

Anno Accademico 2011–2012

To my family, to my friends

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AC	Autonomic Computing
AcOS	Autonomic Operating System
AI	Artificial Intelligence
API	Application Programming Interface
AS	Autonomic System
CFS	Completely Fair Scheduler
CHANGE	Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments
CMP	Chip Multi Processing
CMPs	Chip Multi Processors
CPU	Central Processing Unit
CRAC	computer room air conditioner
CT	Control Theory
DTM	Dynamic Thermal Management
DVFS	Dynamic Voltage and Frequency Scaling
FLOSS	Free, Libre or Open Source Software
GID	Group IDentifier

HRM Heart Rate Monitor

HT Hyper Threading

HVAC Heating, Ventilation and Air Conditioning

I-Cache Instruction-Cache

I/O Input/Output

ILP Instruction Level Parallelism

IT Information Technology

ML Machine Learning

MTTF Mean Time-To-Failure

ODA Observe Decide Act

OS Operating System

PARSEC Princeton Application Repository for Shared-Memory Computers

PID Process IDentifier

PUE Power Usage Effectiveness

QoS Quality of Service

RCSP Resource Constrained Scheduling Problem

RMS Recognition, Mining and Synthesis

SMP Simultaneous Multi Processor

SMT Simultaneous Multi Threading

TAS Thermal-aware scheduling

TDP Thermal Design Power

TID Thread IDentifier

TLP Thread Level Parallelism

Summary

Thermal constraints in high performance computing environments are becoming a major issue for systems' designers which is tackled with different approaches. In addition, the skyrocketing complexity of computing systems of the near future is sought to become unmanageable leaving the question open on how to continue to improve the performance of the computing of tomorrow.

In this work it is proposed an advancement in the context of operating system-based, local Dynamic Thermal Management (DTM) technique deeply rooted in the Autonomic Computing (AC) initiative called ADAPTME, a simultaneously performance and thermal aware scheduler aimed at server environments with general purpose (both batch and interactive) workloads.

After implementing the system as a patch for a commodity open source Operating System (OS), the work is compared to the state of art, resulting in lower average running temperature and slowdowns.

Sommario

I vincoli di natura termica nei sistemi informatici ad alte prestazioni stanno progressivamente diventando un problema per i progettisti di questi sistemi, che cercano di affrontare il problema da diversi punti di vista ed a diversi livelli. Oltretutto, è previsto che la crescita esponenziale della complessità dei sistemi informatici previsti per i prossimi anni a venire diventi rapidamente ingestibile, lasciando aperta la questione su come continuare a migliorare le performance dei sistemi del futuro.

In questo lavoro, che è profondamente legato al mondo dell'Autonomic Computing, proponiamo un miglioramento nel contesto delle tecniche per la gestione locale e dinamica della temperatura realizzata a livello di sistema operativo per mezzo di uno scheduler simultaneamente thermal- e performance-aware orientato a sistemi server con carichi di lavoro generici (sia interattivi che batch): ADAPTME.

Dopo avere implementato il sistema come una patch per un sistema operativo open source largamente disponibile, ci confrontiamo con lo stato dell'arte, risultando migliori sia dal punto di vista della più bassa temperatura media ottenuta che dei ridotti rallentamenti dovuti all'azione del controllo termico.

Chapter 1

Introduction

In this Chapter it is introduced the context and the notions required to motivate the work done in this thesis. In Section 1.1 some major factors that are playing a role in shifting the evolution of modern computing systems towards new paradigms are introduced, and are briefly summarized in Section 1.2. In Section 1.3 some preliminary definitions about the scheduling problem are given, along with the possible goals to reach and a link to the context in which this works is developed.

1.1 Introductory contextualization

The way the semiconductor industry is keeping up in the recent years with the pace of change set by Moore's law [5] has seen a sharp drift from the path that was followed until the early 2000s. As we reach the physical limits of silicon-based transistors miniaturization, the increase of performance of integrated circuits can no longer be obtained by a mere increase of their clock speed, partly due to thermal issues, partly due to issues related to the propagation of the clock signal in the chip's area [6]. The power density of nowadays' microprocessors is higher than ever, potentially causing thermal issues. Moreover, the amount of Instruction Level Parallelism (ILP) that can be extracted from code after years of processors evolution and optimization (deeper pipelines, multiple issues, speculative and out-of-order execution and branch prediction among the others) is

dramatically reduced [7].

In addition, for some years now we are experiencing the explosion of the mobile computing era [6]. With it, we are seeing the exponential increase in the usage of Internet-based and/or social services for the most desperate means: from mobile search and geolocation services to video streaming, from real-time news sharing over social networks to file sharing. The epicenter of our computation is rapidly shifting from the periphery (the terminals which we access the Internet with) to the core of the network (the datacenters within which the largest amount of our computation is actually performed and our data stored) [6]. In addition to this trend, due to the increasing costs of running a datacenter (arisen in recent years in particular due to a general boost of the cost of electricity) many small and medium businesses have seen concrete opportunities in sharing both computing resources and technical personnel among them.

As a result, in the near future we are going to deal with systems whose complexity appears to be approaching the limits of human capability, yet the march toward increased interconnectivity and integration rushes ahead unabated [8, 9]. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure , optimize, maintain, and merge [8, 9].

These considerations led to the conclusion that in order to improve the performance of the computing systems of the future, it was necessary a radical shift in the way they would be designed, both from the hardware and software point of view. At the same time, the term "performance" itself has acquired a connotation ever more frequently tied to the notion of performance with respect to its *cost* and to the *energy* consumed to obtain that amount of computation [2].

This shift marked the beginning of era of the multicore, cloud and autonomic computing (from a hardware and computer systems/software systems point of

view, respectively).

1.2 Contemporary computing: relevant trends and paradigms

In order to cope with the challenges posed by nowadays and future computing systems, new paradigms have been proposed and adopted by major actors in the Information Technology (IT) field and semiconductor industry. In this Section we shall explore two of them: multicore architectures [7] and autonomic computing [8].

1.2.1 Multicore computing

Up to the early 2000s, the increase in the performance of computing systems was mainly obtained by reducing the size of Central Processing Unit (CPU) transistors and consequently by increasing their clock speed [7]. This approach pushed the technology to its limits, in that many factors came into play to limit the possibility to further this approach. The miniaturization process of silicon-based transistors and the consequent increase in power density has reached a point after which it is practically difficult, if not technically impossible, to go further [7, 6]. The main consequences may be summarized as follows:

- the area a given amount of transistors occupy cannot be further reduced,
- a maximum working clock frequency can be identified after which there is at least a critical path over which the signal cannot propagate in time for the circuit to work properly. This is due to a phenomenon known as "clock skew".
- given the minimum area for that amount of transistors, a maximum amount of thermal energy can be dissipated by the packaging using reasonable cooling solutions (this aspect is of particular importance for mobile devices) [6].

Current technological solutions may not drastically overcome these physical limitations. Proposed solutions to improve the performance of computing devices

look at the micro architectural and software levels [6]. As already stated, architectural means to extract ILP have been implemented in the previous years [7]. These solutions yielded greater performance improvements in the past generations of microprocessors, but extracting the residual ILP would give lesser beneficial effects nowadays than in the past [7].

These observations comes at the verge of the explosion of the mobile computing era, where the largest share of computing devices is going to be either embedded or at least mobile (like smartphones or laptops) [6]. Energy efficient solutions capable of providing plenty of computational power are required for the devices of the future, given the gap between battery advancements and smartphones/-mobile devices' capabilities and available computational power [6]. Of particular importance will be how technology will cool the electronics inside these devices, and how to *preventively* and *actively* lower the operating temperature. In this context, a single, general purpose processor rapidly becomes an inefficient solution for delivering the required performances of future devices [6].

The most promising solution to do this set of problems has been identified in the design of multicore systems, CPUs natively capable of running tasks in parallel. These chips realize true parallel execution (in contrast to "perceived" parallel execution typical of traditional single core processors) in that they have multiple so-called *cores* , units capable of independently run the fetch-decode-execute loop [7] (and all the more or less complex stages in the possibly deeply pipelined variations of it). The architectural shift has a number of advantages: we can make smaller, cooler, cheaper CPUs that can perform at least as much as older chips, while using fewer resources, or we can build upon these cores powerful multicore CPUs that can truly run in parallel the tasks of our system. It is foreseen that future CPUs will continue to be focused on improving Thread Level Parallelism (TLP), rather then ILP [7].

Of course, to harness the power offered by these devices, a shift in the programming paradigm is required, too, along with adequate support by the operating system. In the first case, the main problem is to rethink serial code in order to decompose it into a set of parallel tasks, called *threads* [10]. Performance

improvement is obtained by running in parallel the various application threads. From the point of view of the operating system, support should be implemented at different levels, but one component stands among the others: the scheduler.

This component, which is responsible for scheduling the execution of tasks, has been obviously rethought in recent years in the light of CPUs advancements aimed at concurrent execution of multiple tasks. Modern schedulers must take into account more factors than those of pre-multicore era Operating System (OS)s: for example, in order to keep data and instruction cache warm (thus reducing the misses), a modern scheduler is supposed to *pin* (i.e.: to assign to) a given thread to a given core for as long as possible, so as to reduce the thread migration effect and favoring data locality (both spatial and temporal) [10]. Simultaneous Multi Threading (SMT) and fine grained multithreading technology, by which multiple threads' execution is interleaved in the *same* core [7], accentuates the need for an informed and improved OS scheduler.

One peculiar weak point of nowadays multicore architectures is the traffic induced by cache memories. With just a single core accessing cache memory, it is relatively easy to maintain it up to date with main memory: write-back and write-through policies are fairly easy to implement in hardware, and they scale both with main and cache memory dimensions. A different problem arises when considering multi core systems: since two threads of the same application may be accessing shared data, and since these data may be cached, a synchronization protocol must be in place to force coherence among the caches of the cores of the processor [7] (unless adequate support from higher level components is obtained, but this is still a research issue). Snooping based and directory protocols are in place for this very reason, but the burden they impose on the architecture is by no means negligible, as they typically require a communication channel whose bandwidth grows with the number of cores (in the case of snooping based protocols) or significantly added logic to maintain the state of cache data across the distributed memory [7] (in the case of directory-based caches).

Another important factor to take into account from the system designer's perspective is that nowadays computing device should be benchmarked hav-

ing Chip Multi Processing (CMP) and multi threading in mind. This raises the question on how a benchmark should be designed for an effective and extensive coverage of all the relevant factors concurring in a multicore/multi processor environment to yield better performances. Many relevant works in this field have been carried out in the last years (such as Princeton Application Repository for Shared-Memory Computers (PARSEC) [4] or SPLASH-2 [11]). As we will see in Chapter 6, PARSEC 2.1 is a state of the art benchmarking suite typically used in the field to validate both micro architectures (by means of – possibly cycle accurate – simulation) and system software designs (for example, scheduler at OS-level).

Summarizing, the shift is at the architectural, operating system and programming paradigm level: we can obtain the same throughput with less performant but increased number of actual processors, or we can speedup applications by a maximum theoretical limit of the number of truly concurrent threads the processor can run at once. This of course requires the application to be decomposed in parallel tasks, which requires a supplemental effort invested in developing applications. From the point of view of the operating system, we expect it to be capable of efficiently scheduling threads on different cores to better exploit data locality and reduce latency of interactive applications.

1.2.2 Autonomic computing

With the term Autonomic Computing, IBM describes in [9] those systems "[...] capable of running themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them. These autonomic systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there. [...]".

In [9, 12] the authors root the motivations behind Autonomic Computing (AC) stating that "[...] The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of myriad interacting, self-governing components that in turn comprise

large numbers of interacting, autonomous, self-governing components at the next level down. The enormous range in scale, starting with molecular machines within cells and extending to human markets, societies, and the entire world socioeconomy, mirrors that of computing systems, which run from individual devices to the entire Internet. Thus, we believe it will be profitable to seek inspiration in the self-governance of social and economic systems as well as purely biological ones [...]"

In IBM projections for the near future, pervasive computing will drive an exponential growth of the complexity of overall computing systems infrastructure [9, 12]. Their claim is that only if computer-based systems become more "autonomic" – that is, to a large extent self-managing given high-level objectives from administrators – we shall be able to deal with this growing complexity. In [8] the authors identifies a number of sources of complexity in todays systems, and underline the value of AC as a means for putting this complexity under administrator's control.

Cloud computing, as a representative computing paradigm involving highly complex systems, relies on many features of autonomic computing, including many autonomic components. Cloud computing incorporates elements of autonomic computing, since cloud providers would utilize multiple computers and a self-regulating system. Without such measures in place, cloud providers could not keep up with the maintenance costs and demands of the features they provide.

Properties of an Autonomic System (AS)

In IBM's vision of AC, the system must be endowed with a number of characteristics to be called "Autonomic" [8, 9, 12, 9].

"Knowing" itself the system must have detailed knowledge of its components, status, capacity, connections and available resources, either in an exclusive or shared way. This feature is known as "self-awareness",

Self-(re)configuration an AC system must be capable of automatically setting

itself up, given high level administrator policies,

Continuous optimization that is, the system is continuously looking for ways to exploit its resources in the most efficient possible way, monitoring its constituent parts and fine-tune workflow to achieve predetermined system goals,

Self-healing the system must be able to discover problems or potential problems, then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly,

Self-protection it must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity,

Environment knowledge an AS will find and generate rules for how best to interact with neighboring systems. It will tap available resources, even negotiate the use by other systems of its underutilized elements, changing both itself and its environment in the process,

Open world while independent in its ability to manage itself, an autonomic computing system must function in a heterogeneous world and implement open standards,

Predict required resources this is the ultimate goal of autonomic computing: the marshaling of IT resources to shrink the gap between the business or personal goals of our customers, and the IT implementation necessary to achieve those goals without involving the user in that implementation.

The Observe Decide Act (ODA) loop

A recurrent theme in the autonomic computing field is that of decentralized, autonomous control. Self-adaptive systems (i.e., systems employing either self-adaptive hardware or software) rely on control loops to adjust their behavior to internal and environmental changes. Such systems are required to observe themselves and the environment, decide on a sequence of actions to perform, and apply them in order to optimize their operations. The process of observing,

deciding, and acting is customarily referred to as ODA [13]. ODA loop is rep-

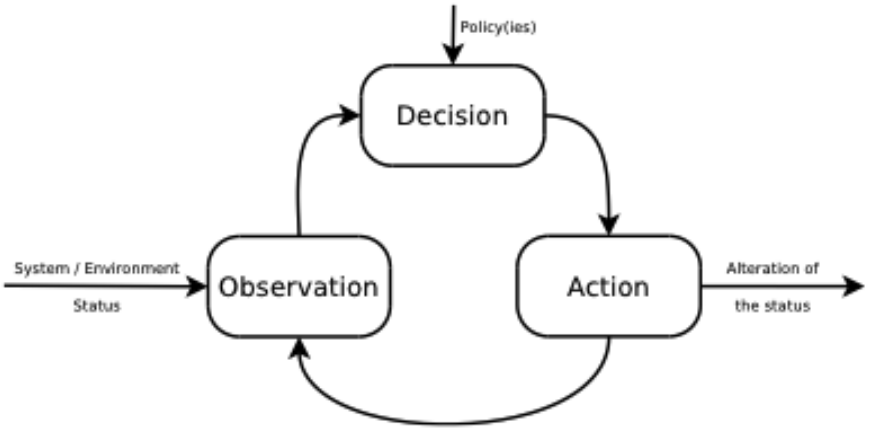

Figure 1.1: The Observe-Decide-Act loop.

resented in Figure 1.1. This representation is the most general autonomic loop scheme and, being the most generic, summarizes their essence. The steps of the ODA loop are

- observation of the internal and environmental status,
- decision of what action (or whether no action at all) is to be taken, based on the observations
- action, i.e. perturbation of the internal or external status in order to modify it towards a better condition for the system.

The observation phase does typically rely on the presence and exploitation of some kind of monitor (for instance, thermometers, throughput meters, latency measures, ...) which is preposed to gather information about the environment and/or the internals of the system. The decision phase takes into account the data gathered through the monitors and an additional input representing the decision policy(ies) implemented in the system, which can be based on different techniques. The action phase is performed through actuators, which are devices (virtual or physical) that allow the system to alter its internal status or the operating environment [3].

The ODA loop is the minimal representation for the class of control loops that can be used to equip a system with self-adaptive properties; in an autonomic sys-

tem, the ODA control loop may appear at different levels, where each component of the system is thusly controlled at a lower level and there is a higher-level controller that orchestrates the modules towards the specified goals [3].

Summarizing, AC is an emerging field of IT aimed at increasing the degree of automation and autonomy of tomorrows computing systems, elements of which are already implemented in nowadays data centers. The claimed capability of self-governance and self-optimization, in particular, are interesting in the light of exponentially increasing complexity of tomorrows computing systems.

1.2.3 Power efficient computing

Power and energy are increasingly becoming prominent factor when designing the full spectrum of computing solutions, from supercomputers and data centers to handheld phones and other mobile or embedded computers [6]. Research is currently focused on managing power and improving energy efficiency of today and tomorrow computing devices. In fact, power density has become one of the major constraints on attainable processor performance.

With respect to mobile and embedded devices, this translates directly into how long the battery lasts under typical usage [6]. The battery is often the largest and heaviest component of the system, so improved battery life implies smaller and lighter devices [6] or added functionalities available in the device.

Power and energy considerations are at least as important for devices connected to a power supply. The electricity consumption of computing equipment in a typical U.S. household runs to several hundred dollars per year [6]. This cost is vastly multiplied in business enterprises: an analysis made by IT analysis firm IDC estimates the worldwide spending on power management for enterprises was likely in the order of magnitude of 40 billion $ in 2009 [6].

Being efficient at consuming power has three main advantages. The most obvious one is that reduced power consumption directly implies reduced running costs. Second, reduced power consumption leads to less complex designs of power supplies, power distribution grids and backup units, that reduces the costs to the whole infrastructure. Last, since reduced power consumption implies

reduced heat generation, those costs associated to heat management are reduced [6].

Thermal management, in particular, is becoming increasingly important due to the level of miniaturization of modern electronics and the increased blades density typical of modern data centers. Increased compaction (such as in future predicted blade servers) will increase power densities by an order of magnitude within the next decade, and the increased densities will start hitting the physical limits of practical air-cooled solutions [6, 14]. Studies, most notably concerning servers and hard-disk failures, have shown that running electronics at temperatures that exceed their operational range can lead to significant degradation of reliability, i.e. they experience exponentially reduced Mean Time-To-Failure (MTTF) values [15]. The Uptime Institute, an industry organization that tracks data-center trends, has identified a 50% increased chance of server failure per each 10°C increase over the 20°C range [6, 16, 17]; similar statistics have also been shown over hard-disk lifetimes [15, 18, 19, 17]. Temperature directly affects also power consumption, clock latency and since processor leakage power increases exponentially with temperature, also CPU power consumption [20, 21]. At 90-nm-process nodes, leakage accounts for 25 to 40% of total power consumed [20]. At 65-nm-processes, leakage accounts for 50 to 70% of total power absorbed [20]. Moreover, a 15°C increase in temperature might causes signal propagation delay of approximately 10 to 15% [20]. Processor cooling is also a significant problem for mobile devices as thermal conditions can affect user experience through both heat dissipation and potentially intrusive cooling [22].

For large computing systems like supercomputers and data centers, the costs for running Heating, Ventilation and Air Conditioning (HVAC) systems for temperature management can be estimated as more or less an additional dollar spent for every dollar spent on electricity [6, 14]. Up to 80% of data center construction cost is attributable to power and cooling infrastructure [1, 14], and chiller power, a historically dominant data center energy overhead, scales quadratically with the amount of heat extracted [14]. Research is ongoing in alternate cooling technologies (such as efficient liquid cooling), but it will still be important to be

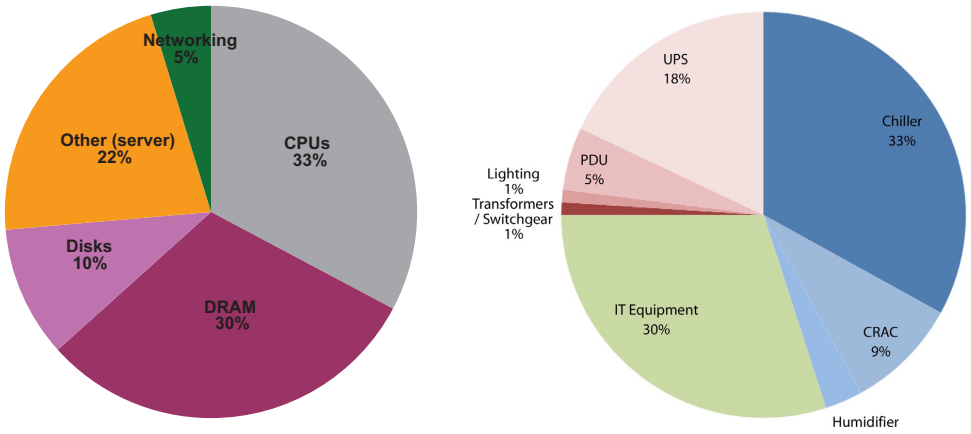

Figure 1.2: Breakdown of total data center hardware and overheads costs of a representative Google datacenter [1].

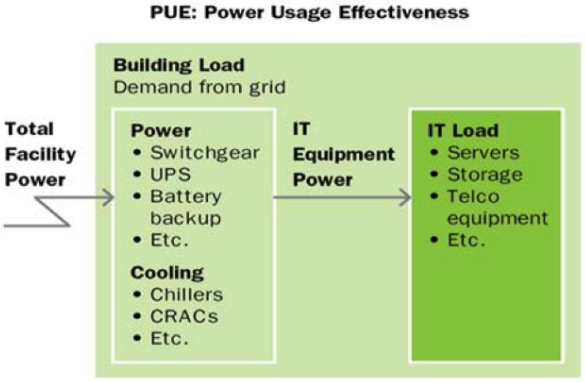

Figure 1.3: The model for PUE [2].

efficient about generating heat in the first place [6].

In order to capture these overheads in a metric, the Green Grid, a non-profit IT organization that addresses power and cooling requirements for datacenters and the entire information service delivery ecosystem, defined the Power Usage Effectiveness (PUE) [2]. PUE is defined as the total facility power/IT equipment power, effectively measuring a form of overall data-center infrastructure efficiency; refer to Figure 1.3.

Power management issues are only expected to be more and more predominant in the foreseeable future [6]. On the mobile devices side, the gap between advances in battery capacity and reliability and the ever growing increases in mobile-devices functionalities will become a major limiting factor for the devel-

opment of the entire mobile/embedded industry [6]. New battery technologies (such as fuel cells or graphene-based capacitors) might mitigate it, but designing more power-efficient systems will still be the main driver for full battery capacity exploitation. Tethered devices are affected, too: data from the U.S. Environment Protection Agency points to steadily increasing costs for electricity [23]. For data centers, recent reports highlight a growing concern with computer-energy consumption and show how current trends could make energy a dominant factor in the total cost of ownership [24] up to the point at which power and cooling cost might overtake hardware costs [25, 24, 26].

In Chapter 2 we explore the techniques that have been developed in recent years in order to deal with the problems just exposed, both at low- (micro architectural/electronics) and high-level (operating systems, scheduling algorithms, datacenters management techniques).

1.3 Scheduler properties and goals

In modern, multiprogrammed computing environments, as well as larger computing systems like data centers, it is frequent to have multiple running processes or threads competing for CPU time. This situation occurs whenever two or more processes or threads (*tasks*) are in ready state [10]. Depending on the number of available processing elements (which is variable from one in a legacy unicore machine to 4 in modern commodity desktops to tens of thousands on contemporary data centers), the scheduling process must occur so as to decide what task to run, where. The part of the OS that takes care of managing this process is called *scheduler* [10].

1.3.1 Preliminary definitions

In order to better understand the following section, we shall give here some preliminary definitions of interest for the scheduling problem [10].

The process is a basic concept for multiprogramming operating systems, as it defines the basic structure for managing code in execution. A process is funda-

mentally a container that holds all the information needed to run a program [10]. Processes are one of the oldest and most important abstractions that operating systems provide. They support the ability to have (pseudo) concurrent operation even when there is only one CPU available [10].

Process. *A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the OS, a process may be made up of multiple threads of execution that execute instructions concurrently.*

Multiprogramming. *The OS characteristic to have several programs in memory at once, each in its own memory partition, and the rapid switching back and forth between them.*

The switching between a program and the other is one important feature of the scheduling algorithm. There are mainly two ways by which processes are scheduled and de-scheduled on CPUs:

cooperative A nonpreemptive (cooperative) scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU,

preemptive preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler [10].

The time that is assigned a process for running is called "quantum". On most systems this is a fixed amount of time (typically specified at compile-time) but in others is a run-time variable (typically to improve the interactivity of the system).

Moreover, depending on the progress of the execution and on the current request being serviced, processes can be in one of a number of *states*. Even if different operating systems have different states for representing (maybe slightly) different processes' situations, here are reported the three most commonly found ones:

running the process in this state is currently running on one processing element

ready this state signals that the process is ready to be assigned to a CPU

blocked the process is waiting for some condition to happen before becoming runnable

Depending on the finite state automaton that describes the states and transitions of the scheduling algorithm, we may have a more fine-grained control over the states of a process.

In traditional operating systems, each process has an address space and a single thread of control (that is almost the definition of a process). Nevertheless, there frequently are situations in which it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space). This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work. Moreover, since creating and destroying threads is much faster than it is for processes, applications that creates and destroys a large number of threads during their execution will experience a substantial speedup[10, 27]. Most importantly, threads are useful in systems with multiple CPUs, so as to achieve true parallelism.

Depending on the threading model we may have different definitions of threads. In particular, there are systems (such as Linux) that blur the line between processes and threads, and others that don't. This definition applies to the traditional threading model where threads are different entities than processes.

Thread. *A thread of execution (or, simply, thread) is a sub-entity within a process; it is a specific part of the executing program in charge of doing some kind of elaboration. A process contains several threads which share the address space, open files and, in general, the resources assigned to the process.*

Multithreading. *The OS characteristic to have several threads running really in parallel. This of course requires hardware support.*

1.3.2 Problem statement

As we already exposed in Section 1.3, scheduling is the activity of choosing which process is going to be run in the next quantum of CPU time. This is a matter of interest for this thesis since we are going to investigate an innovative scheduling technique aimed at simultaneous optimization of both temperature and applications' performances.

We recall here one possible, and very general, definition of Resource Constrained Scheduling Problem (RCSP), given in [3, 28]. This definition refers to a generic problem 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)*.

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

In the context of computing task scheduling, we may identify as *resources*, for example, CPUs, Input/Output (I/O) devices and buses, *activities* as processes and threads of processes and, as possible *objective function*, the minimization of the total execution time of the activities.

1.3.3 Batch, interactive and real-time scheduling

With respect to the objective function that has to be minimized in the RCSP, a brief introduction on the characterization of the typologies of workloads is necessary. Depending on the kind of workload of the system, different scheduling policies may be implemented in order to reach different goals [10, 27].

Traditionally, a suggested classification of the possible workloads environments is the following [10, 27]

Batch In batch systems, there are no users waiting for a quick response to a short request. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves throughput, which is a major goal of these systems.

Interactive these activities have a certain degree of interactivity with users. This is the typical case of applications running in desktop computers. Preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is needed to prevent this behavior.

Real-time These workloads are characterized by having to respect a specific deadline for doing their job. In systems with real-time constraints, preemption 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. This category is traditionally further divided up into:

Soft Real-time the deadlines of these loads are not strict, which means that the system can tolerate that some tasks do not complete in time; the system is said to be working in a *best-effort* manner.

Hard Real-time the deadlines for these activities are strict, which means that in case the scheduler cannot guarantee their execution by the expressed deadline, the system should return an error.

Another classification useful for this context is that of scheduling goals, that depend on the system's workload and are obtained by appropriate policies [3, 10, 27].

fairness by fairness we mean the attitude of a scheduler to assign an equal amount of resources to all the processes

balance a balanced system is one that exploits at its best the resources available; it tries to keep all the resources as busy as possible

throughput maximization the scheduler tries to complete the maximum number of tasks per unit of time

turnaround time minimization by turnaround time we mean the total difference of time between the beginning of the job and its end; a scheduler may try to minimize the average turnaround time for the set of scheduled jobs

response time minimization a scheduler may try to reduce the time between a user request and its service

proportionality differing from fairness because the system tries to assign a fair share to each *user*, instead of each *activity*

deadlines meeting the scheduler enforces the meeting of the deadlines

predictability/deterministic behavior the scheduler must say in advance if the deadlines expressed may be met or not.

A recent development in the context of scheduling algorithms is the introduction of knowledge about the status of the system's temperature in the scheduler algorithm, in order to try to find jobs schedules and system settings compatible with dynamic thermal constraints [29, 30, 31, 32]. This allows for a new goal to be introduced, namely

thermally constrained by which it is indicated the property by which the scheduler computes a schedule of jobs that keeps temperature under a given set point.

Those schedulers that aim at achieving this goal belong to the Thermal-aware scheduling (TAS) category. As we will better described through Chapters 3 and 4, our work is a major development in the context of thermal-aware schedulers.

1.4 Conclusions

We have introduced a number of concepts and motivations that are relevant to this thesis. A more thorough and comprehensive overview of Dynamic Thermal Management (DTM) techniques will be given in Chapter 2, along with the status of art of scheduling in mainstream Free, Libre or Open Source Software (FLOSS) operating systems. In Chapter 3 the Autonomic Operating System (AcOS) vision is introduced. In Chapter 4 it is introduced the basic techniques and the general ideas underlying the work. In Chapter 5, the system that has been designed and implemented will be dedescribed in great detail, and its experimental evaluation reported in Chapter 6.

Chapter 2

Related works

In this chapter it is explored the state of art of some major technologies, techniques and systems which are relevant to this thesis. In Section 2.1 two important Free, Libre or Open Source Software (FLOSS) Operating System (OS) are introduced along with a brief history of schedulers implemented in the latest iterations of their development. Focus is put in particular on the scheduler presented in Section 2.1.2, since this is the basic work on which it will be developed part of the thesis contribution. After this, some of the most relevant problems in terms of power consumption in computing environments are introduced, along with ways to reduce them. In particular, the focus is on Dynamic Thermal Management (DTM) techniques, both at the hardware and at the software level. In the conclusion it is described a state of art OS self-aware component, namely Heart Rate Monitor (HRM), which will be the major means by which the envisioned system acquire knowledge about itself.

2.1 Overview of major Free or Open Source schedulers

To date, there are many available FLOSS operating systems for the most disparate usages. Keeping in mind the difference between the OS and its kernel (the latter is a part of the former but there are parts of the OS that are not part of the kernel), in this section focus is on the two most relevant works in this context, GNU/Linux and FreeBSD.

2.1.1 Linux schedulers

Linux is a Unix-like monolithic kernel developed by the open source movement since 1991 [33]. It is the kernel most widely installed in open source operating systems [33]. Nearly all major GNU distributions employ Linux as kernel. At the moment, the guide of the development of Linux is directed by his first developer, Linus Torvalds, and the Linux Foundation.

Early Linux Schedulers

The scheduler portions of the kernel has undergone a lot of development since its inception, in 1991. Early Linux schedulers used minimal designs, not yet focused on massive architectures with many processors or even Simultaneous Multi Threading (SMT) capabilties [34]. The 1.2 Linux scheduler used a simple and fast circular queue for runnable task management that operated with a round-robin scheduling policy [34].

Linux version 2.2 introduced the idea of scheduling classes which is now a common feature of general purpose scheduling infrastructures, permitting differing scheduling policies for real-time tasks, non-preemptible tasks, and non-real-time tasks. The 2.2 scheduler also included support for Simultaneous Multi Processor (SMP) [34].

The 2.4 kernel included a relatively simple scheduler that operated in $O(N)$ time (as it iterated over every task during a scheduling event). The 2.4 scheduler divided time into *epochs*, and within each epoch, every task was allowed to execute up to its time slice. If a task did not use all of its time slice, then half of the remaining time slice was added to the new time slice to allow it to execute longer in the next epoch. The scheduler would simply iterate over the tasks, applying a goodness function (metric) to determine which task to execute next. The weak points of this approach are the relatively inefficiency, limited scalability, and overall weakness for real-time systems. It also lacked features to exploit new hardware architectures such as multi-core processors [34].

The $O(1)$ scheduler

To overcome the limitations of the 2.4 scheduler, O(1) was designed and introduced in 2.6. The scheduler was not required to iterate the entire task list to identify the next task to schedule (resulting in its name, O(1), which means that the scheduling decision takes constant time, however the number of tasks to iterate over). The O(1) scheduler kept track of runnable tasks in a run queue (actually, two run queues for each priority level-one for active and one for expired tasks), which meant that to identify the task to execute next, the scheduler simply needed to dequeue the next task off the specific active per-priority run queue. The O(1) scheduler was much more scalable and incorporated interactivity metrics with numerous heuristics to determine whether tasks were I/O-bound or processor-bound [34].

On fundamental problem with O(1) scheduler became the large mass of code needed to calculate heuristics, which was difficult to manage and lacked algorithmic substance. the change came in the way of a kernel patch from Con Kolivas, with his Rotating Staircase Deadline Scheduler (RSDL), which included his earlier work on the staircase scheduler. The result of this work was a simply designed scheduler that incorporated fairness with bounded latency. Kolivas' scheduler impressed many (with calls to incorporate it into the current 2.6.21 mainline kernel), so it was clear that a scheduler change was on the way [34].

The CFS scheduler

The main idea behind the Completely Fair Scheduler (CFS) is to maintain balance (fairness) in providing processor time to running tasks [34]. This means processes should be given a fair amount of the processor. When the time for tasks is out of balance (meaning that one or more tasks are not given a fair amount of time relative to others), then those out-of-balance tasks should be given time to execute.

To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the *virtual runtime*. When the virtual runtime is "low" (relatively low) it means that the amount of time a task has been permitted

access to the processor has been "low", and viceversa. The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it [34].

Rather than maintaining tasks in a run queue, CFS maintains a *time-ordered red-black tree*. A red-black tree is a tree with two important properties. First, it's self-balancing, so no path in the tree will ever be more than twice as long as any other [34]. Second, operations on the tree occur in O(log n) time in the number of nodes in the tree. Insertion and deletion are quick and efficient [34].

With tasks stored in the time-ordered red-black tree, tasks with the lowest virtual runtime are stored toward the left side of the tree, and tasks with the highest virtual runtimes are stored toward the right side of the tree. The scheduler then, in order to achieve fairness, picks the left-most node of the red-black tree to schedule next. The task accounts for its time with the Central Processing Unit (CPU) by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable. In this way, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to maintain fairness. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks [34].

CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay. The decay factor states how fast the virtual runtime changes in time, allowing for more o less cpu time to be accorded to tasks. That's an elegant solution to avoid maintaining run queues per priority [34].

Another interesting aspect of CFS is the concept of group scheduling, another way to bring fairness to scheduling, in particular in the face of tasks that spawn many other tasks. Also introduced with CFS is the idea of scheduling classes, by which task belongs to a scheduling class, which determines how a task will be scheduled. A scheduling class defines a common set of functions that define the behavior of the scheduler [34].

To date, CFS is the default Linux scheduler.

2.1.2 FreeBSD schedulers

FreeBSD is one of the most fortunate descendant of BSD [35, 36] and, to date, one of the major FLOSS kernels available. Many other kernels, notably Apple's Darwin, derive from this project.

The 4.4BSD scheduler

FreeBSD inherited the traditional BSD scheduler when it branched off from 4.3BSD. 4.4BSD is the default scheduler available in FreeBSD up to version 5.1, included [35]. FreeBSD extended the original scheduler's functionality, adding scheduling classes and basic SMP support, without twisting its fundamental foundation. Two new classes, real-time and idle, were added early on in FreeBSD. . It was initially designed for uniprocessor systems, but with the advent of multicore architectures, it was adapted to support SMP and SMT technology (like Intel Hyper Threading (HT)) [35].

The FreeBSD time-share-scheduling algorithm is based on *multilevel feedback queues*. The system adjusts the priority of a thread dynamically to reflect resource requirements (e.g., being blocked awaiting an event) and the amount of CPU time consumed by the thread. Threads are moved between run queues based on changes in their scheduling priority (hence the word "feedback" in the name "multilevel feedback queue"). Whenever a thread other than the currently running one attains a higher priority and if the current thread is in user mode, the system switches to that immediately. Otherwise, the system switches to the higher-priority thread as soon as the current one exits the kernel citedesign-freebsd. The system tailors this *short-term scheduling algorithm* to favor interactive jobs by raising the scheduling priority of threads that are blocked waiting for Input/Output (I/O) for one or more seconds and by lowering the priority of threads that execute for a significant amounts of time.

The priority of a thread is determined by two values associated with its thread structure: `kg_estcpu` and `kg_nice`. The value of `kg_estcpu` provides an estimate

of the recent CPU utilization of the thread by means of a so called *exponential decay filter*. The value of kg_nice, instead, is a user-settable factor that ranges between -20 and 20, so that it is possible to selectively allow some processes more CPU time than others, based on administrator's needs. The normal value for kg_nice is 0. Negative values increase a thread's priority, whereas positive values decrease its priority [35]. A thread's user-mode scheduling priority is calculated after every four clock ticks (typically 40 milliseconds, this quantity being defined at compile time) by running this equation:

$$\mathtt{kg\_user\_pri} = \mathtt{PRI\_MIN\_TIMESHARE} + \left\lceil \frac{\mathtt{kg\_estcpu}}{4} \right\rceil + 2 \times \mathtt{kg\_nice} \quad (2.1)$$

PRI_MIN_TIMESHARE indicates the minimum value attainable by a thread before going into kernel or real-time mode. Values lower than that and greater than PRI_MAX_TIMESHARE.

Three *positive* terms are summed up to define the priority: the first is the lowest priority value a thread in TIME_SHARE class can have, the second is the decay factor, which depends on how long the thread has been running on the CPU, and the last one is the user-settable value that helps administrators in biasing the behavior of the scheduler [35].

kg_estcpu is the term that estimates the usage of the cpu by the thread [35]. It is computed by means of an *exponential decay filter*, a mechanism by which the system forgets of about 90 percent of the CPU usage accumulated in a 1-second interval over a period of time that is dependent on the system load average [35]. This quantity is computed via the equation

$$\mathtt{kg\_estcpu} = \frac{2 \times \mathtt{load}}{2 \times \mathtt{load} + 1} \times \mathtt{kg\_estcpu} + \mathtt{kg\_nice}$$

where the load is a sampled average of the sum of the lengths of the run queue and of the short-term sleep queue over the previous 1-minute interval of system operation [35]. In case the thread was preempted and put to sleep waiting for an

event to occur, he decay filter value is computed only once at thread wakeup as:

$$\texttt{kg\_estcpu} = \left[\frac{2 \times \texttt{load}}{2 \times \texttt{load} + 1}\right]^{\texttt{kg\_slptime}} \times \texttt{kg\_estcpu} \tag{2.2}$$

where $\texttt{kg\_slptime}$ is a value incremented by 1 for every second the thread is in sleeping or stopped state.

Additionally, idle priority threads are only run when there are no time sharing or real-time threads to run[35]. Real-time threads are allowed to run until they block or until a higher priority task is placed onto the multilevel feedback queue [35].

This scheduler has been later superseded by ULE in order to overcome its limitations [35].

The ULE scheduler

ULE has been developed to overcome the main limitations of 4.4BSD, which may be summarized as:

- being developed in the late 90s, support for SMP and SMT has not been included in the scheduler at design time, but only as an incremental patch; support to these micro architectural features is limited [35],
- for analogous reasons, no or very limited support for affinity is present [35],
- the algorithm is O(N) in the number of schedulable threads [35].

We refer to *affinity* to describe a scheduler that only migrates threads when necessary to give an idle processor something to do.

2.2 Policies for energy efficiency

There is a wide spectrum of techniques that allow for power efficient computing to take place [22]. In this section focus will be on *local* techniques only, i.e. those that can improve power efficiency of a single machine, in contrast to *global* techniques, which are aimed at cluster, data centers and, in general, groups

of cooperating machines. The latter differ from the formers because they typically take into account the spatial disposition of the machines (like the "hot aisle/cold aisle" displacement [16]), the way Heating, Ventilation and Air Conditioning (HVAC) and computer room air conditioner (CRAC)s installed and job scheduling techniques that span across the entire data-center. Local techniques are particularly interesting for the autonomic computing community, since decentralized optimization and control is one of the fundamental *tenets* of the Autonomic Computing Manifesto [9].

Local techniques fall in two different and complementary categories: those that are implemented at the microarchitectural level [21] and those implemented at software level (either operating system level or programming language/paradigm level) [37]. They belong to the first category techniques such as Dynamic Voltage and Frequency Scaling (DVFS), fetch throttling, and clock gating [38], which are standard features of modern microprocessors [22]. These techniques are particularly suitable for *reactively* reducing cores temperature, and reduce the burden of worst-case temperature management [39, 40, 41]. They don't try to *proactively* contrast the rise of the temperature. Software-level scheduling schemes [29, 42, 43], instead, proactively take into account thermal management and the temperature/performance trade-offs, and belong to the second class. Hybrids techniques between the two categories, such as HybDTM [44], and programming paradigms/models such as GREENSOFT [45], complete the overview of currently available techniques. In the next section, focus will be put on *software-level* techniques, being of primary interest for this work.

2.3 Dynamic Thermal Management techniques

In recent years, as technology for microprocessors is entering the nanometer regime, the power densities of microprocessors have doubled every nearly two years [46, 47]. This increase in power densities has led to two major problems. Firstly, high energy consumption is a limitation for mobile, battery operated devices. Secondly, higher temperatures directly affect reliability and cool-

ing costs, both for battery-operated and tethered devices. Unfortunately, cooling techniques for these devices must be designed to cope with the *maximum* possible power dissipation of the microprocessor, even if it rarely occurs, in typical applications, that critical temperatures (due to continuous maximum power usage) are reached. On one hand, worst-case dynamic thermal management avoids performance degradation while failing to provide a proper control over temperature; on the other hand, preventive dynamic thermal management introduces performance degradation while providing a proper control. In addition to this, failures may happen to CRAC unit, or CPU fans upsetting the thermal environment in a matter of minutes or even seconds. Rapid response strategies, often faster than what is possible at a facilities level, are required to cope with these (infrequent, yet not impossible) situations [43]. On average, cooling techniques are an overkill solution, yet they are necessary to cope with critical temperature spikes. This situation is only expected to be more and more of an issue, given current and expected levels of transistors miniaturization and thermal packing availability.

A lot of effort is being put into finding finding ways lo limit the negative side effects that overheating has on computing devices. The main motivations behind this are the growing power densities of current and foreseen computing systems, the ever growing electricity costs (which impact on the air conditioning costs of the computing environment) the consequently increasing direct costs of HVACs and CRACs in modern data centers along with indirect costs occurring due to reduced lifetime and reliability of computing electronics. For high-performance Chip Multi Processing (CMP)s, thermal control has become an important issue due to their high heat dissipation [29]. Thermal packaging, fans, CRACs and HVACs are the primary solution, but suffer from high cost and complexity, apart from their being designed for worst case thermal conditions. Therefore, DTM techniques have been getting more popular for their low cost, flexibility [29] and stated goal of allowing designers to focus on average case rather than worst-case thermal conditions [39].

For these reasons, as already discussed, it is increasingly important to man-

age, if not limit, energy consumption and temperature in current and future computing systems. Dynamic thermal management techniques have undergone a lot of development in the recent years due to the need of limiting the ever growing operating temperature of modern multicores processors.

Scientific research has been focused in the recent years on developing two main lines of work: the first one studies DTM techniques aimed at micro architectures and low-level electronics, while the second focuses on the operating system and in particular on thermal-aware scheduling policies, both for multicore/multiprocessor machines and whole data centers.

In the next section is given a review some major DTM techniques that are studied (and some are readily available in nowadays chips) in the context of hardware based DTM. In section 2.3.2, instead, focus will be put on solutions that rely on OS support for thermal control.

2.3.1 Hardware Dynamic Thermal Management

Under this category fall all those techniques that apply any form of thermal/energy or power control at the architectural/electronics level.

The first DTM techniques that were put into play used to be simple mechanisms aimed at guaranteeing that a thermally overloaded system would not break down due to insufficient cooling; they were simple hardware solutions to solely limit peak temperatures. In recent years, these techniques evolved into energy and power saving, thermal control mechanisms commonly found in readily available CPUs. These schemes do typically throttle performance to lower power consumption when a preset temperature threshold is reached [29].

Microarchitecture-related DTM relevant researches are [39, 48, 49, 50, 51, 52, 21, 53, 54, 55].

Dynamic Voltage Frequency Scaling

One of the most common DTM technologies implemented in nowadays microprocessors is DVFS. As the name suggests, this mircoarchitectural-level DTM technique dynamically varies the voltage and the operating frequency of the mi-

croprocessor so as to find a point in the configuration space that allows the system to reach a suitable thermal and power saving condition [43].

DVFS dynamically chooses the best tradeoff between power consumption and performance selecting a stable voltage supply/working frequency pair configuration [39, 56, 57, 55]. Of course, since dynamic power dissipation is quadratically linked to switching frequency and linearly linked to voltage, lowering one or the other or both directly reduces power consumption and heating [58, 48].

Research has traditionally focused on single core architectures [52, 21, 53], even though in recent years we are assisting to a shift of interest towards multicore ones. One major limiting factor is that in nowadays architectures in not always available a per-core possibility to select the set point [59].

Notably, in [60], this technique has been employed in combination with a thermal-aware operating system, resulting in a hybrid solution between hardware and software DTM where the resulting system can (thanks to a thermal model of the cpu and power profiles of programs) maximize processor usage under varying conditions, while implementing an optimal policy for DVFS usage. DVFS has the major drawback of impacting in a non-discriminatory way on all the applications running in the system [39].

Clock gating

This technique allows a processor in low power mode to disable some clock propagation paths in large portions of the circuit. Switching off the clock eliminates the dynamic power leaving only static power. During this time the core or chip (depending on the number of voltage domains) is slowing down the total processing time increases, but in return the temperature is dropping [61, 55, 39].

Speculative execution throttling

Speculative execution is a mechanism by which microprocessors try to keep the pipeline as full as possible by issuing and executing instructions belonging to parts of code that *may* execute in case the branches these instructions belong to are effectively taken [7]. This implies that unless a perfect branch predictor is

in effect (which is of course an ideal, and not real, device) some instructions will have to be issued and executed, but will not commit. All in all, this Instruction Level Parallelism (ILP) mechanism increases the overall performance of the system, but at the price of a waste of power that may not be negligible [49, 39].

Speculative execution throttling turns off this microprocessor's feature so as to limit to the strictly necessary the number of instructions that are going to be executed (and committed), at the price of performance reduction [56].

Instruction-Cache (I-Cache) throttling

I-Cache throttling allows the processor's fetch bandwidth to be reduced when the CPU reaches a temperature limit [39]. Again, this kind of technique reduces the number of instructions executed per second, limiting the number of instructions that may be issued, on average, every clock cycle,with obvious impacts on performance.

2.3.2 Software Dynamic Thermal Management

In this category fall all those DTM techniques that are implemented at a higher level than the micro architectural/electronics one.

The main motivation for the existence of a different class of mechanisms, is that the main limit of hardware DTM is that is only suited to *reactive* responses, that is, typical of emergency situations like those related to failure of cooling systems and analogous situations where thermal loads, due to peak activity of overloaded systems, are not effectively disposed of. Moreover, since at such low level there is very limited if not at all knowledge about OS tasks, these techniques affect in a non discriminatory way those tasks that are effectively heating the system as well as those that are not. Finally, since the typical hardware response is throttling, a severe performance degradation for a class of applications that demand high performance is likely [29].

Software based techniques plays a fundamental roles in both these aspects. First of all, they tend to be *proactive*, in that they try to *prevent* heating, in the first place, to be generated in excess of the disposal capacity of the system. Moreover,

they can finely discriminate which tasks are effectively heating the systems and which tasks are not. This allows for a sensible increase in average performance with respect to the employment of purely hardware solutions

In this way, they successfully achieve an average reduction of temperature for the entire runtime of the system at a reduced cost.

Idle cycle injection

Due to the dependence between leakage power and temperature, different distributions of idle time will lead to different temperature distributions and, consequentially, energy consumption. Idle cycle injection has been recently implemented by [62, 22] as a means to lowering CPU temperatures. In [63], the authors address the problem of distributing idle time among different tasks at different voltage and frequency levels for energy minimization. In their work, the authors assume a processor model having two basic operational modes: active and idle. Idling the processor activates the low power mode and obviously lowers the temperature, while keeping it active increases it. Since executing a `nop` equivalent instruction results in putting the processor in idle mode, injecting a varying number of these instructions in the processor results in an overall lowering of the temperature (which can be adjusted according to a given goal). Injecting idle instructions, obviously, involves a tradeoff between application performance (intended as execution time) and maximum temperature reached. Apart from this tradeoff, another major drawback is a low selectivity of the slowed down processes.

Core migration

Core migration [42], is a multicore-aware strategy by which threads are run on different cores in order not to incur in penalties due to idling while at the same time distributing heat in a more homogeneous way on CPU's die. As already pointed out in subsection 1.2.1, one of the major challenges for operating systems schedulers aimed at multicore architectures is to find a way to achieve maximum parallelism while preserving as possible data locality in caches; obvi-

ously, migrating threads is the worst way for obtaining locality, thus this technique is potentially associated with a low cache hit rate side effect.

2.3.3 Thermal Aware Scheduling

The main idea behind this kind of scheduling, which is operated at OS-level, is to execute jobs in such a way to induce variations in CPU temperature [39]. Thermal-aware scheduling (TAS) realizes a kind of scheduling that has the explicit goal of keeping system's temperature below a given threshold [43, 31]. By means of an intelligent schedule, overall system temperature can be put under control. The actual schedule may comprise a set of parallel actions to be taken during schedule execution (like, for example, idle cycle injection or switching DVFS set point).

This typically involves classifying running tasks as "hot" or "cold" , depending on their relative degree of CPU-boundedness, I/O-boundedness and interactivity level [64]. This classification allows the scheduler to choose when and where (i.e.: on which core or socket, depending on the architecture) to physically execute that task. This decision can be based on a power/thermal prediction model [43, 60, 30, 49, 64, 31], TAS-specific heuristics [43, 64], optimal policies possibly obtained by means of approximate solutions [65, 30, 60, 66], task-related performance counters [67, 64], physical location of the machine [68] or CPU [69]. Moreover, depending on the field of application, TAS may come as an *online* or *offline* scheduler. In the first case, research is typically focused on everyday computing or data centers which are subject to substantially varying, unpredictable workloads [69, 68, 22], and the scheduling problem is analyzed in the light of *soft-real time* scheduling problem. Offline TAS schedulers, instead, are typically targeted at *hard-real time* platforms [70, 71], as they typically employ tasks schedules which are known in advance of execution.

Since our work continues this direction of research, in the following paragraphs it is given a description of these decision factors so as to better compare our work to the state of art.

Power profiles of applications and thermal prediction models

Power is dissipated inside a processor in many different ways. From pure code execution to memory access to static power leakage, energy may be employed for a number of different purposes. In recent years, due to the increasing interest around the problem of workload characterization, many different techniques and tools have been developed to model power profiles of applications and thermal behavior of microprocessors. Applications' power profiling becomes an appealing feature to systems designers interested in developing DTM techniques, and in particular TAS, when this knowledge is coupled with that of the CPU micro architecture, since this allows for the development of thermal predictive models based on task execution.

Authors in [64] use a simple thermal model for characterizing the application, based on the work by [72] where each task is assumed to reach a steady state temperature and maintain it until its ending. This simplification allows for a simple representation of thermal contribution and simplifies the TAS problem.

In [43], authors consider a simplified thermal model for a single core processor (stating that this is easily scalable to chip multiprocessors devices) based on [73] called "dynamic compact thermal model". Even though they admit some oversimplification, they claim to be able to predict a temperature violation for the entire die in a timely fashion. At the same time, they find a good tradeoff between accuracy and computational burden (in terms of memory and time) for their online task power estimator, which they claim relies only on the last available reading of task power usage for prediction purposes. Accuracy for such an estimator is in the order of 10%, which is comparable to other online power profilers found in the literature.

Authors in [30] restrict their focus on the set of *batch*, lowly-interacting workloads in the context of soft real-time. They rely on readily available tools such as Wattch [74] and SimpleScalar [75] for power profiling applications. After profiling a set of batch jobs, they conclude that since the variance of the job's temperature between different assigned quanta is low, three main phases can be identified (start, steady state and shut down) and the central one (the steady state) is rep-

resentative of the thermal behavior of the application, i.e. the thermal effect that a core is going to experience when the task is run. The current thermal profile is given as input to the scheduler along with the set of runnable tasks, and their expected power consumption/thermal behavior. A look up on pre-computed look up tables stored inside kernel memory allows the scheduler to efficiently take a decision in order not to violate the thermal constraint. If the deadline of the task is missed, the task is discarded.

In practically all works that are based upon a thermal model of the CPU, the authors have used either ATMI [76] or HotSpot [77] as a thermal modeling framework.

Heuristic, formal and approximate solution approach to TAS

Since multiprogrammed/multitasking operating systems continuously switch between processes in order to give the illusion of parallelism, scheduling decision must be taken in a timely fashion by the operating system. For example, FreeBSD takes a scheduling decision every 40 ms [36]. It is clear how much important it is to have an efficient scheduling algorithm, since it will be very frequently called during the execution of the system.

As authors in [78] claim, under their definition of TAS, the policy to find an optimal temperature-aware schedule has NP-Hard complexity. For this reason, optimal TAS schedules may not be computed in those contexts where the arrival of the tasks is not known in advance, since rescheduling of these tasks would rapidly become infeasible.

Research has been focused on two different approaches. The first is the study of optimal scheduling algorithms for hard real-time workloads, where it is supposed that tasks are recurrent and known in advance, along with their strict deadlines. The most relevant work in this line of research is reported in [78] and in [60], where authors first define what an optimal thermal-aware policy does and then derive the complexity for the algorithms required for that policy to be optimal. After concluding that the complexity of the policy is NP-Hard, they propose two different approaches to TAS. The first is to implement an offline sched-

uler which solves a complex dynamic programming problem that finds the best schedule for a set of hard real-time workloads given their deadlines, their recurring arrival order and the thermal model of the CPU. The second is to implement a heuristic used to approximately solve the scheduling problem, assuming that the arrival order is not known in advance. Even if it is not optimal, *a posteriori* simulations demonstrate that this heuristic provides good results anyway.

The second line of research is based on less formal approaches, based upon the knowledge about the thermal model of the CPU and the power profiles of the applications, and is more frequently found in the literature.

One such heuristic is Power Based Thread Migration [79], where the cores are sorted by their current temperatures (increasing) and tasks are sorted by their power dissipation numbers (decreasing). At the beginning of every migration interval, task i is mapped to core i according to their respective lists, i.e. the highest power dissipating task is assigned to the coldest core and the least power dissipating task to the hottest core [60].

In another work, [43], the authors rely on power profiles of applications to determine an ordering between tasks that is considered as an effective way to keep temperature low. The heuristic that they use to order tasks is based upon the observation that if we call x and y two tasks, and x is hotter than y, then if we schedule y before x the final resulting temperature will be lower than if we scheduled y and then x. This heuristic, which they called ThresHot [43], performs better than MinTemp [32], another heuristic that schedules tasks in such a way that the coolest and the hottest tasks are scheduled whenever temperature falls outside of the specified thresholds (on a per-CPU basis). The claim is that both performs better, i.e. better mitigate temperature, than a simpler heuristic which simply lowers that priority of those processes that are causing more heating.

The idle cycle injection technique has been thoroughly explored in [22]. In that work, the authors built a system based on a probabilistic model for injecting idle cycles of variable length for controlling heating and providing responsive, inexpensive, fine-grained control, allowing individual threads to absorb substantial portions of the burden of cooling. This work focuses on reducing average-case

processor operating temperatures, exploring the trade-offs between application performance and long-term thermal behavior through preventive thermal management.

2.4 Heart Rate Monitor

In the context of Autonomic Computing (AC), many systems have been implemented in order to realize the notion of "knowledge of self". Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) research team has developed HRM a flexible and efficient monitoring infrastructure. The ideas behind Heart Rate Monitor (HRM) resemble those at the base of Application Heartbeats and exploit the well-known idea of *heartbeat*, already used in the past for measuring performance and signaling both progresses and availability [80]. Application Heartbeats was born as a simple, usable, and portable user-space active monitor. However, when it comes down to functionality, the great portability of Application Heartbeats becomes a weak spot. 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 share 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 applications and makes use of synchronization even for signaling progresses. HRM is an active monitor, integrated with Linux and FreeBSD 7.2, 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 exposes a compact API, allowing applications and system developers to instrument applications and build both user- and kernel-space adaptation policies. This interaction model between applications and adaptation policies, mediated by the API, can be seen as a producer/consumer model in which applications work as producers and adaptation policies work as consumers.

2.4.1 Definitions

In this section it is provided a set of general and specific definitions to better understand the remainder of the section. The focus will be on the FreeBSD porting of HRM, although the majority of the definitions still applies to both FreeBSD and Linux.

A running instance of a program, including both the code and the data, is called a process. In FreeBSD, a unique Process IDentifier (PID) identifies a process. A thread conceptually exists within a process and shares both the code and the data with the other threads of a given process. In FreeBSD, a unique Thread IDentifier (TID) identifies a thread. A task is any unit of execution, being it either a process or a thread. Given these definitions, an application can be defined as a set of tasks pursuing a set of objectives (e.g., encoding an audio/video stream). Being a set of tasks, an application can be either single-threaded, multi-threaded, multi-processed, or any feasible combination of them; HRM accounts for any feasible composition of these entities.

A *heartbeat* is a signal emitted by any of the application's tasks at a certain point in the code and is a metaphor for some kind of progress. For example, it has been used as a measure of throughput, as a measure of latency and a measure of contention. When heartbeats are employed for throughput means, a *hotspot* is a performance-relevant portion of code executed by any of the applications tasks; a hotspot 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, it is useful to define the concept of *group*; a group is a subset of applications tasks pursuing a common objective (e.g., encoding a video stream in audio/video encoder). 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 applications 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 relation 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*.

Chapter 3

Autonomic Operating Systems

In this Chapter the thesis proposal for the extension of an autonomic computing systems is presented, from a high level vision of the approach to the details regarding the proposed methodology to extend a commodity operating system with an autonomic layer. The high level vision is illustrated in Section 3.1, where the goals, the model and the components comprising this vision are defined. The focus is then moved, in subection 3.1.1, to the direct contributions of this thesis to the creation of an autonomic layer in a commodity Operating System (OS) provided with a monitoring facility to which it is added a proof of concept of a thermal and performance aware adaptation policy.

3.1 High Level Vision

The work proposed in this thesis builds upon the ideas expressed by the autonomic computing community [8] in order to decline them into a feasible approach to the realization of that vision. The driving motivation for this effort comes from the observations, in part already exposed in subsection1.2.2, regarding current and foreseen complexity of computing systems, too prone at exposing their complexity to software developers. Since computer architectures lack a strong support for evolution (e.g., multi- and many-cores and heterogeneous computing units) applications developers are required to take into account ever lower-level details about the target architectures, making the software design

process more and more complex. This issue can be addressed with a neat distinction between system and applications developers, where the former are in charge of supporting the computing architectures and offering suitable high level interfaces to the latter.

Within this context, the overall research objective aims at the creation of self-aware computing systems to deal with this complexity. Any kind of modern computing device would benefit from this advancement: from mobile devices and desktops to servers to mainframes and huge computing facilities. This vision is shared with the Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) research group, whose work is deeply rooted in the autonomic computing field, founded on the belief that system developers should employ autonomic computing techniques for enabling computing systems to continuously adapt in face of evolution of systems. Within this context, the concept of performance is extended beyond the mere idea that the faster, the better, but it comes to include objectives such as minimization of power consumption and thermal efficiency together with the goal of ensuring to the users an experience as close as possible to their needs [3].

CHANGE team proposes an approach to the realization of this vision built upon the fundamental concept of Observe Decide Act (ODA) control loop (see Section 1.2.2), implemented at different levels within the envisioned system. Starting from the architectural level, each component can benefit from the presence of a local ODA loop to realize autonomic management "in the small"; then, to higher levels, broader ODA loops should orchestrate the different subsystems and, at the top level, a system-wide control loop, aware of the system as a whole, should be in charge of pursuing maximum runtime performance, meant as in the broader sense explained above.

A major role would be played by Artificial Intelligence (AI), Control Theory (CT) and Machine Learning (ML) techniques [3], since the orchestration of autonomic components must be dealt with also (and most importantly) in the context of dynamically changing environments – such as those foreseen for the near future.

3.1.1 Thesis contribution

The long term goal of the CHANGE group [81] is the realization of a heterogeneous autonomic computing system such as that introduced in Section 1.2.2, obtained by creating methodologies and designs for computing systems able to adapt their behavior according to their internal and environmental status.

To achieve this goal, the group works on various aspects of computing systems, from architectures to operating systems and development tools. In order to realize the vision where application developers must only focus on what their applications have to do, leaving all the architecture-dependent details to the autonomic features of the systems where they will be deployed, all of the components of a computing system should be modified in order to create an autonomic behavior in the system *as a whole* [3].

Given this premises, the first and most important computing system component to be redesigned – possibly from scratch – in an autonomic direction is the OS.

The claim is supported by at least three evidences:

- The only means for applications to interact with the external environment and access resources in modern computing environments is by means of requests directed towards the system layer which exposes the system resources towards the applications which is part of the OS; hence, it has a direct link with the applications, which are the entities that the autonomic system must serve according to their performance requirements.

- on the other hand, the OS has direct access to the hardware resources and it is in charge of managing them.

- Since the OS is a software system, it is possible to work at this level in an agile way, without the need of requiring hardware modifications to the architectures or to the components. This could be a further step to improve the autonomic features once the autonomic base system in the OS layer will be ready [3].

Being the glue between the hardware and the applications, the OS is the best candidate for the introduction of a first autonomic layer inside the system. This layer will serve as the basis and support for successive improvements both at architectural and applicative level. Within this context, the contributions of this thesis are:

- The theoretical definition of a set of autonomic policies implemented at OS scheduler-level for simultaneous performance and thermal aware scheduling.
- The porting of the Heart Rate Monitor (HRM) monitoring infrastructure to a commodity operating system, namely FreeBSD, and the implementation of both the thermal and the performance aware policies in the same scheduler.
- The evaluation and characterization of the implemented system with real workloads

3.2 Autonomic Computing Model and Components

Most modern computing systems can be subdivided into three layers: *Hardware Components*, *Operating System* and *Applications* [3]; the proposed model for autonomic computing augments it:

- Within the hardware layer, each component should embed an integrated autonomic ODA-based controller to autonomously manage its lower level parameters in order to maintain a stable working status compatible with global user's goals.
- At the upper level, applications should embed similar software mechanisms to tune their behavior, taking into account users' preferences.
- Acting as a "glue" , the OS should be aware of the presence of autonomic components both at the hardware and the application level. Hardware and software should expose informations about their status towards the OS

which should in turn use these data to implement a number of ODA loops aimed at altering the runtime status according to users' needs.

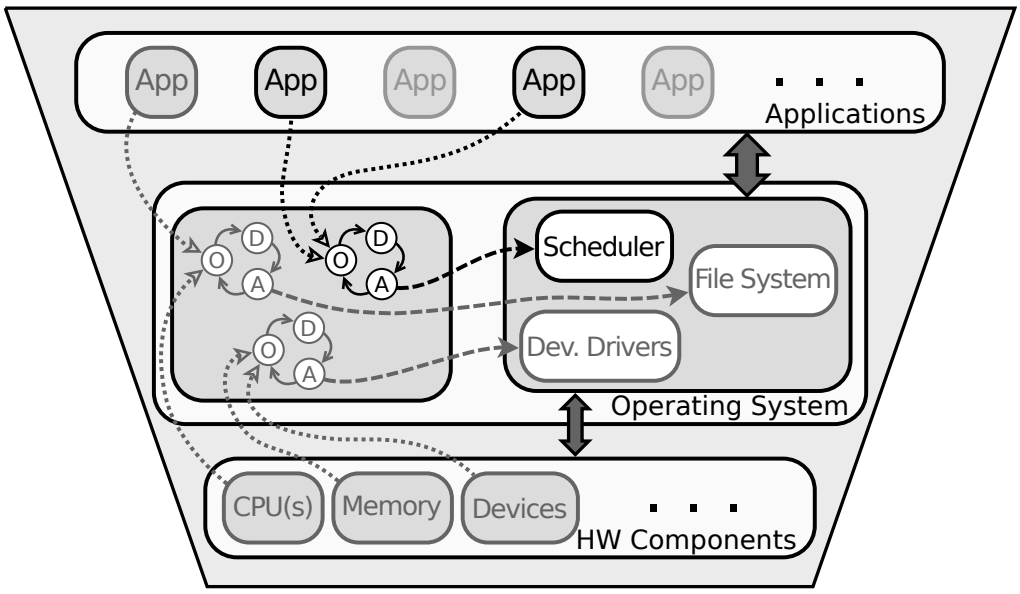

Figure 3.1: The vision of an Autonomic System: ODA loops are integrated in the system at various levels, so as to allow each layer to act according to user's needs with her direct intervention (*courtesy of Davide Basilio Bartolini, [3]*)

As argued in subsection 3.1.1, the first place where to act in realizing the autonomic vision and extending the classic computing system is the OS; a pictorial representation of this vision is given in Figure 3.1 which shows how ODA loops are integrated in the form of local autonomic controllers between the autonomic layer, the applications and the hardware. Here, the extension of the classic three-layered structure of a computing system is represented by the autonomic layer which has been added into the OS.

In this vision, the autonomic layer allows applications to explicitly communicate their performance goals (which can be specified by the developers or the users) while is continuously monitoring them to capture any deviation from the required performance. On the hardware side, components are monitored both in terms of performance and in terms of health status (for instance, working temperature, voltage, power consumption etc...) and their performance capabilities and health status are considered with respect to the current and desired condi-

tions. Lastly, the OS components' behavior is guided by user's goals and runtime system constraints.

This kind of control requires an actual ODA loop to be implemented. The implementation of the following components is proposed

- *monitors*: these components realize the notion of *Observe*
- *adaptation policies*: these components realize the notion of *Observation* and *Act*.

The monitor is in charge of collection informations between users on one side and the autonomic layer on the other. It is characterized by what measure it records (called the target measure) and must also provide a means of specifying which are the desired values for its target measure. For example, talking about a performance monitor, the expectation is to provide a means by which stating applications' performance goals in terms of a range of desired values in the metric used by the monitor that represent the desired runtime state for each application. This range of desired values is the *goal* for that target measure.

The adaptation policies access the information provided by one or more monitors and elaborate on them to *act* in one or more possibly contrasting ways on the system. The decision mechanism may be based on different kinds of techniques: from machine learning to control theory or any feasible heuristic. Adaptation policies determine whether any corrective action is needed whenever the measures on their observed monitors do not match the goals specified by an application. In this case, it can act on one or more actuation hooks within the system where it can modify some parameters (e.g. the clock frequency of a processor or the Central Processing Unit (CPU) time assigned to an application) to alter the system status. These hooks are frequently referred to as "knobs". In this way, each adaptation policy, coupled with one or possibly more monitors, identifies a separate ODA loop within the autonomic system.

Since within the same system different adaptation policies may coexist and have contrasting goals or clash in the use of actuation hooks, there is the need for an higher level component in charge of coordinating the operation of the adap-

tation policies. This component, called adaptation manager, has the role of coordinating the autonomic action having access to all and the adaptation policies and the monitoring informations in the system. The main task of the adaptation manager is to selectively enable or disable adaptation policies with the aim of reaching the global system goals. For instance, as it is in Figure 3.2, there could be different adaptation policies working on the same target and using the same monitoring information but different decision mechanisms (e.g., an heuristic versus a control theory-based policy); in this case, the adaptation manager would be in charge of choosing the best policy according to the runtime context.

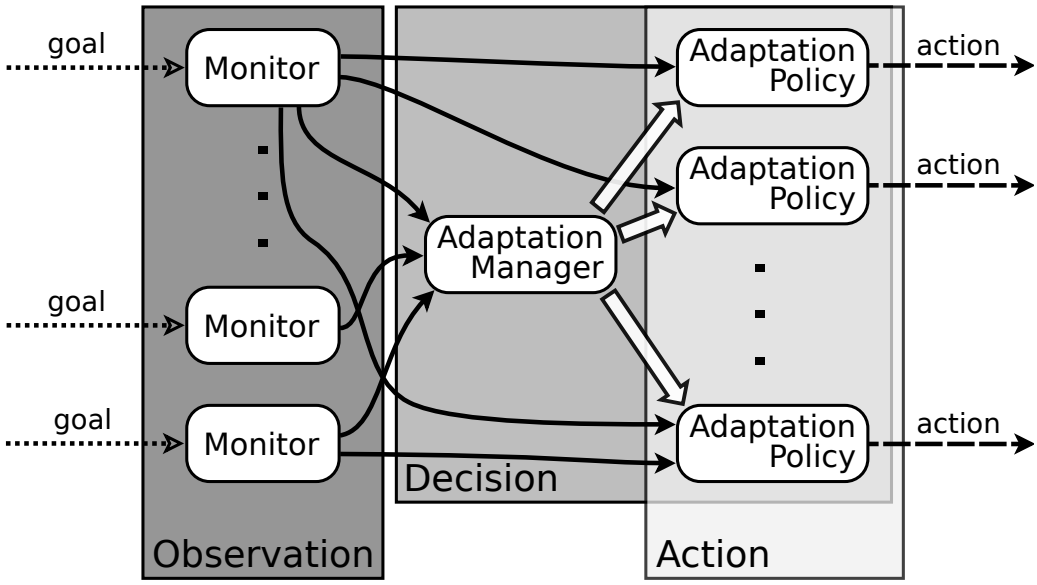

Figure 3.2: The adaptation manager. This component interacts with different monitors and adaptation policies in order to realize user's high level goals. (*courtesy of Davide Basilio Bartolini*, [3]

3.3 An Autonomic Operating System extension: ADAPTME

As explained in Section 3.1, for the autonomic vision to be materialized into a working system the main needed component is the OS. The first choice faced was whether it was better to design an OS from scratch or extend an existing one. Due to the limited amount of time on one hand, and due to the amount of readily available, high-quality OS code, it was decided to extend an existing commodity operating system, FreeBSD. This particular choice is due to two factors

- [3] and [82] have already implemented some autonomic components and functionalities into Linux; in their work, the authors develops a state of art monitor called HRM of which it is made a porting to a different OS to demonstrate its portability,
- [22] implements their system in FreeBSD; since one stated goal of this work is to compare an autonomic system exposing similar functionalities to theirs, the choice of the OS is constrained.

Their work is the base system that it is further extended in the sense of the envisioned autonomic operating system: ADAPTME. In this thesis we focus on the implementation of two different adaption policies and on the porting of a state of art monitor in order to demonstrate the feasibility of the elaborated approach, leaving space for further incremental changes at the monitoring, at the adaptation policies and the adaptation manager layers.

Our aim is to design, implement and validate a system in which two different (and potentially adversary) goals are reached by the system based on high levels indications of the user. In this work we assume a scenario in which the system administrator is interested in reducing the average running temperature of the system, while penalizing as little as possible applications while allowing them to meet the expected Quality of Service (QoS). In order to do this, we assume that the applications have been modified so as to signal ADAPTME about their current progress and desired goal (in a similar manner to what has been described in the vision proposed in this Chapter) and that the system can be configured to specify the maximum average desired temperature, i.e. that temperature under which that the system should stay under fully loaded conditions for a reasonably long period of time.

Our first extension to the previous body of work related to Autonomic Operating System (AcOS) is the porting of HRM to FreeBSD, in order to give "self-awareness" to the system. This allows us to introduce two adaptive policies, namely the *thermal-aware* and the *performance-aware* policies. The first one is aimed at scheduling management meant for putting a cap on average running temperature by means of idle cycle injections. The second one is aimed at management

of performance of instrumented applications by means of priority recomputation influenced by current and desired heart rates. Both policies are based upon a control theoretical framework that allows us to deduce some properties of these two policies.

Chapter 4

Proposed Methodology

In the following Sections it is given a detailed descriptionn the theoretical aspects of this work. In section 4.2, a brief dissertation about the control theoretical aspects of this work and its applications is given. Heart Rate Monitor (HRM) is then described in section 2.4 as a means for the autonomic system to "know itself". The two concurring policies composing our Thermal-aware scheduling (TAS) algorithm are presented in 4.3.1. A brief recall on the practical aspects of this work is then given in 7.2.

4.1 Motivation

As we have seen in Chapter 2, Thermal Aware Scheduling (TAS) is an interesting solution to the increasing problem of heat control in modern computing environments. After having explored a number of local TAS techniques it was decided to further develop the state of art, implementing an advanced TAS *performance-aware* policy in Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE)'s Autonomic Operating System (AcOS). By TAS performance-aware policy it is meant the property by which the scheduler does its best to guarantee that the temperature will stay under a given set point under any workload, *selectively* penalizing the performance of those tasks that are exceeding their heart rate (see 2.4) or those that have not expressed any goal. In case it is not possible to satisfy the goal of all the HRM-enalbled tasks without

violation of the thermal constraint, the scheduler will inject idle time prevalently during the quanta of non HRM-enabled processes but also during the quanta of the others. This allows to selectively give more Central Processing Unit (CPU) time (and the consequent possibility to generate heat) to those applications that are of interest to the system administrator, while allowing him/her to satisfy the temperature constraint.

This feature can be regarded as an autonomic feature, in that the administrator has to give only high level indications to the system about the expected performance and maximum temperature allowed, and the the system employs a form of self-regulation that realizes the expressed goals.

Among the different techniques that are employed at software level (and in particular at scheduling level), it was decided to focus on *idle cycle injection*. Different reasons led us to this decision:

- this technique is practically agnostic to the underlying architecture, since `nop` or `hlt` instructions are commonly implemented in nowadays processors
- this technique assumes no particular thermal model of the CPU, nor power profiles of the applications; our scheduler can consequently re-compute priorities based upon a relatively compact model and fast calculation. This is to keep up with a possibly large amount of scheduling decisions, making it an ideal choice for overloaded systems. Obtained results make us feel confident about the quality of the performed control, since temperature is effectively kept under control and performance objectives met.
- most recent TAS works (such as [22]) implemented with success this technique on commodity operating systems, making them a good benchmark for comparing our solution to the state-of-art.

Typical scheduling infrastructures in commodity operating systems follow to the race-to-idle principle: applications are run to completion in order to idle the system as soon as possible, thus increasing applications' throughput and decreasing their latency. This is true for both interactive and batch workloads.

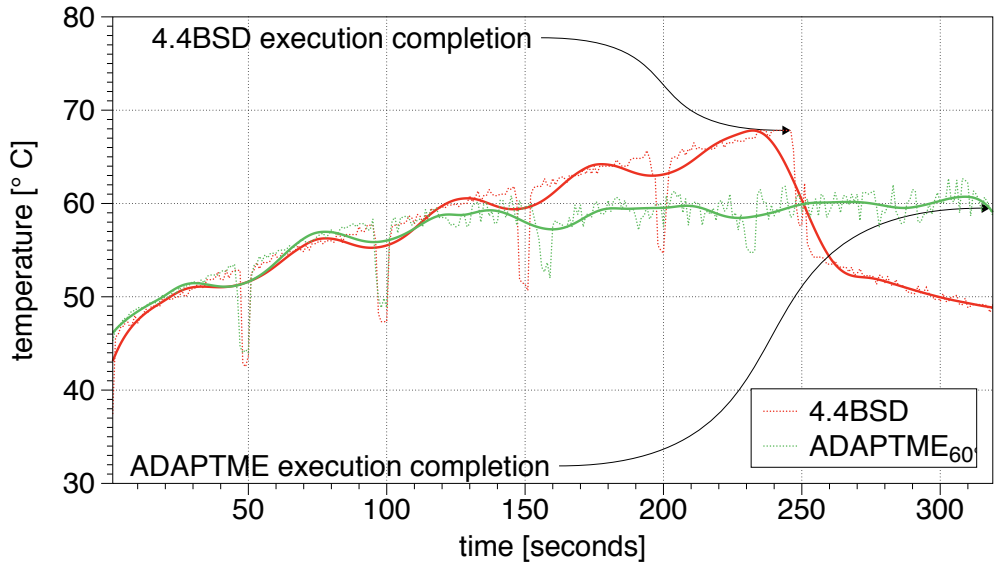

Figure 4.1: Race-to-idle versus thermal aware approach. In the graph it is easily seen how the execution under ADAPTME with a thermal constraint of 60°C of our benchmark application results in a longer total run-time but lower average temperature. On the other hand, pure 4.4BSD, race-to-idle execution completes more rapidly but involves a not negligible difference in running peak temperature (in this experiment more than 8°C).

While this behavior has been traditionally considered optimal, nowadays and future computing environments benefit from added considerations regarding thermal constraints. Our approach is based on the fact that, if applications can afford a decrease in their throughput or increase in their latency, the scheduling infrastructure may exploit policies to produce less heat and thus, on average, to lower the average running temperature and power consumption. As already pointed out in section 1.2.3, lowering the average temperature greatly reduces spending on cooling infrastructure (e.g., fans, air-conditioners) and greatly improves the mean average life of electronic devices. Figure 4.1 shows the difference between the race-to-idle approach and the thermal-aware one.

Various state of the art approaches presented in Chapter 2 slow indiscriminately down processes in order to obtain temperature reduction, and whenever they do not, they do not have an explicit mechanism by which applications can signal what their minimum acceptable Quality of Service (QoS) level is. Even though this allows to cool down the system, not all the applications can afford a throughput decrease or latency increase. In soft real-time systems, applications

provide deadlines and quality of service to notify their goals and constraints; it is believes that a similar approach can be applied to desktop and server systems, allowing users to provide applications performance goals and applications to signal execution progresses or latencies.

Two different polices are implemented, one thermal-aware and one performance-aware, the combination of which results in ADAPTME, a novel thermal and performance aware scheduling infrastructure based on a popular general purpose scheduler, the 4.4BSD scheduler, implemented in the FreeBSD Operating Sysem. Although it was tried not to introduce significant changes in the scheduling infrastructure, it was observed that the performance-aware policy would have possible caused system instability (for example: task starvation): for this reason, it was implemented the policy based on notions of Control Theory so as to tackle this problem from a stability point of view. In the next section an outline of the reasoning is given to the reader.

4.2 Control Theoretical thermal and performance aware policies

It is now introduced a thermal-aware and a performance-aware policy extending the scheduling infrastructure of a commodity operating system, FreeBSD 7.2.

These two policies execute during different times of the scheduler activity and have different priorities. In fact, the thermal-aware policy is activated at each context switch, while the effects of the performance-aware policy do so only during priority recomputation (which occurs, by default, every 40 ms).

In Section 4.3 it is better described how the control theoretical models are effectively translated into a working mechanism; for now, it is only introduced the theoretical framework.

This work is inspired by the thermal-unaware scheduler by [22]. Conversely to that work, it has been adopted a thermal-aware control-theoretical mechanism in place of a thermal-unaware probabilistic mechanism to achieve temperature

control at scheduling level.

Here follows a formal definition of the activity of the scheduler.

The performance-aware policy makes use of applications' performance goals (i.e., user-specified throughput metrics) to adapt threads priorities by increasing or decreasing the amount of CPU time threads have assigned. These performance goals are expressed by means of a flexible performance goal infrastructure implemented at Operating System (OS)-level called HRM. It will be more thoroughly described the main features of this component in section 2.4. It also features a control-theoretical mechanism to drive applications priority; the policy requires users applications to signal performance goals and applications progresses. It is to distinguish between legacy and non-legacy applications according to whether they do or do not allow specifying performance goals and progresses by means of HRM.

Denoting the performance of the i-th application measured at time k as $r_i(k)$ (the "heart rate"), this time, the target of the controller is to take action so as the performance of every application does not decrease under the performance goal r_0; as already stated, this is a "best-effort" policy, since there is no guarantee about the outcome of the control. This is due to the fact that our model recomputes priorities on a per-task basis, and does not take into account all running tasks at once.

The model of the system is assumed to be the one reported in Equation 4.1.

$$r_i(k+1) = r_i(k) + \eta_{i,j}\Delta priority_{i,j}(k) \tag{4.1}$$

Intuitively, the priority of the task at time instant $k+1$ depends on the priority given at the previous time instant and on a performance increase or decrease dependent on the difference between the current goal and the current heart rate. Even thought it was possible to take into account more variables to better describe system, it was found that under the conditions of the experiments we describe in Section 6 this model works well enough.

$\Delta priority_{i,j}(k)$ is the priority of the j-th thread of the i-th application and

spans the interval between -50 and 50 while $\eta_{i,j}$ is an unknown parameter. This bound on the control action allows for a more fine grained control over tasks' priority, as we will see in the description of the original 4.4BSD scheduler, in Section 2.1.2 . The control-theoretical system calculates $\Delta priority_{i,j}(k)$ per thread per application at each sampling instant. Also in this case, the closed-loop system is designed to be a pure delay, which means that after one step of the controller execution, the set point r_0 is transferred to the output rate. The priority will be increased if and only if the performance risks to be too high, while it will be lowered whenever the performance is lower than the performance goal. The $\eta_{i,j}$ value cannot be given a priori due to its dependence on the workload of the machine, therefore the regulator is coupled with an adaptive estimator that updates the value of the expected $\eta_{i,j}$ per thread per application.

The thermal-aware policy features another control-theoretical mechanism. Defined the temperature of the i-th processing core measured at time k as $T_i(k)$, the target of the controller is to act so as the temperature of the processing core does not exceeds the value T_t; the model of the system is assumed to be the one reported in Equation 4.2.

$$T_i(k+1) = T_i(k) + \mu_i \cdot idle_i(k) \tag{4.2}$$

Intuitively, the temperature of the i-th core at time instant $k+1$ depends on the temperature at the previous time instant and on the amount of time that the system idles. Even thought it was possible to take into account more variables to better describe system, we found that under the conditions of the experiments we describe in Section 6 this model works well enough.

$idle_i(k)$ is the percentage of idle time injected in the i-th core in the time interval between the k-th and $k+1$-th sampling instant and spans in the interval between 0% and 100% of $sched\_quantum$, the quantum of time assigned to tasks in execution. μ_i is an unknown parameter. The control-theoretical system, designed as an adaptive deadbeat controller [83], computes $idle_i(k)$ per core at each sampling instant. A *deadbeat controller* is a controller synthesized so as the

closed-loop transfer function equals a pure delay (z^{-1}), which means that after one step of the controller execution, the set point T is transferred to the output temperature via the controller and the system transfer functions. It is possible to analytically demonstrate that, if μ_i is known, then the set point signal will be attained and the temperature will be kept below the reference level [83] [ma questo e' vero a prescindere dal sistema S di riferimento?]. The intuitive behavior of this controller is that idle cycles will be injected if and only if the temperature risks to be too high, while the control strategy will output 0 whenever there is no possibility of exceeding the reference value. Whenever the μ_i value cannot be given a priori, however, it needs to be estimated based on the current execution on the machine, therefore the deadbeat controller is coupled with an adaptive component that updates the value of the estimation of μ_i per core, based on the last measurements, in an autoregressive modeling fashion. This is of course our situation, since a fixed value for μ_i would imply knowledge about the dynamics of the workloads, and is not an acceptable solution.

4.2.1 Derivation of priority update equation

Here it is derived the equation of the deadbeat controller governing the task's priority update.

The derivation assumes the presence of two components, namely the system under control and its performance controller. This is the same theoretical framework of 4.4BSD scheduler, where the system is the operating system with all the running tasks and the scheduler is the priority update controller.

Here follows the derivation of the equations that yield the transfer function of an adaptive deadbeat controller; i.e. a controller whose property is to enforce the closed-loop transfer function to equal to a pure delay,meaning that after one step of the controller execution, the set point P_0 is transferred to the output performance P_i.

First of all, let's assume a model as in Figure 4.2.

It was decided to keep our model simple enough for the scheduler to keep up with a large number of scheduling decisions per each decision phase, so a naïve

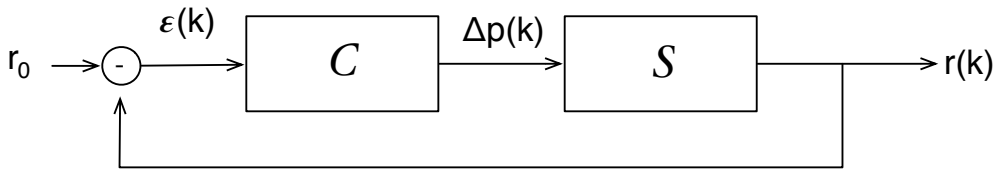

Figure 4.2: The setting of the control problem

representation of the priority of task P at step i, P_i, given target heart rate r_0 and current heart rater$_i$ will be

$$S : r(k+1) = r(k) + \mu\Delta p(k) \tag{4.3}$$

Now the delay operator is made explicit and is factored out

$$\begin{aligned} z \cdot r(k) &= r(k) + \mu\Delta p(k) \\ (z-1) \cdot r(z) &= \mu\Delta p(k) \\ S : \frac{r(z)}{\Delta p(k)} &= \frac{\mu}{z-1} \end{aligned}$$

Then, it is extracted the basic equation for the loop transfer function and constrained it to be the unitary delay operator:

$$\begin{aligned} loop : \frac{C \cdot S}{1 + C \cdot S} &= \frac{1}{z} \\ \frac{C \cdot \frac{\mu}{z-1}}{1 + C \cdot \frac{\mu}{z-1}} &= \frac{1}{z} \\ C \cdot \mu &= \frac{z - 1 + C \cdot \mu}{z} \\ C \cdot \mu \cdot z - C \cdot \mu &= z - 1 \\ C \cdot \mu \cdot \cancel{(z-1)} &= \cancel{z-1} \\ C \cdot \mu &= 1 \\ C &= \frac{1}{\mu} \end{aligned}$$

Finally, the derivation of the equation for $\Delta p(k)$ from the previous steps:

$$\Delta p : \Delta p(k) = C \cdot \epsilon(k)$$

$$\epsilon(k) = r_0 - r(k)$$

$$\Delta p(k) = \frac{1}{\mu} \cdot (r_0 - r(k)) \tag{4.4}$$

In this way, the transfer function from r_0 to $r(k)$ becomes:

$$\begin{aligned} r(k+1) &= r(k) + \mu \cdot \Delta p(k) \\ r(k+1) &= r(k) + \mu \cdot \frac{1}{\mu} \cdot (r_0 - r(k)) \\ r(k+1) &= \cancel{r(k)} + r_0 - \cancel{r(k)} \\ r(k+1) &= r_0 \end{aligned}$$

Which is exactly the definition of deadbeat controller. The $\Delta p(k)$ amount is to be summed as the third term of the priority update equation reported in Equation 2.1; the resulting priority update equation results in:

$$kg\_user\_pri = PRI\_MIN\_TIMESHARE + \left\lceil \frac{kg\_estcpu}{4} \right\rceil + 2 \times kg\_nice + \Delta p(k) \tag{4.5}$$

4.2.2 Derivation of idle-time injection equation

In an analogous fashion, here it is derived the equation for the percentage amount of idle time of a given core:

$$S : t(k+1) = t(k) + \mu \cdot idle(k) \tag{4.6}$$

Now the delay operator is explicited and factored out

$$\begin{aligned} z \cdot t(k) &= t(k) + \mu \cdot idle(k) \\ (z-1) \cdot t(z) &= \mu \cdot idle(k) \\ S : \frac{t(z)}{idle(k)} &= \frac{\mu}{z-1} \end{aligned}$$

Then, it is extracted the basic equation for the loop transfer function and constrained it to be the unitary delay operator:

$$\begin{aligned}
\text{loop} : \frac{C \cdot S}{1 + C \cdot S} &= \frac{1}{z} \\
\frac{C \cdot \frac{\mu}{z-1}}{1 + C \cdot \frac{\mu}{z-1}} &= \frac{1}{z} \\
C \cdot \mu &= \frac{z - 1 + C \cdot \mu}{z} \\
C \cdot \mu \cdot z - C \cdot \mu &= z - 1 \\
C \cdot \mu \cdot \cancel{(z-1)} &= \cancel{z-1} \\
C \cdot \mu &= 1 \\
C &= \frac{1}{\mu}
\end{aligned}$$

Finally, the derivation of the equation for $idle(k)$ from the previous steps:

$$\begin{aligned}
idle(k) : idle(k) &= C \cdot \epsilon(k) \\
\epsilon(k) &= t_0 - t(k) \\
idle(k) &= \frac{1}{\mu} \cdot (t_0 - t(k))
\end{aligned}$$

In this way, the transfer function from t_0 to $t(k)$ becomes:

$$\begin{aligned}
t(k+1) &= t(k) + \mu \cdot idle(k) \\
t(k+1) &= t(k) + \mu \cdot \cancel{\frac{1}{\mu}} \cdot (t_0 - t(k)) \\
t(k+1) &= \cancel{t(k)} + t_0 - \cancel{t(k)} \\
t(k+1) &= t_0
\end{aligned}$$

Since $idle(k)$ has to be a percentage of the quantum of time, we rewrite it in this way:

$$idle(k) = \frac{idle(k)}{quantum} \cdot 100$$

4.3 Autonomic policies

In this section it is explored the relationships that incur between the two autonomic policies so as to describe how do they accomplish thermal mitigation and performance awareness.

4.3.1 Thermal-aware policy

The thermal aware policy is in charge of adjusting the amount of time a given core idles during a quantum of time. After each iteration of the control loop, it is indicated the system for how much of the following quantum of time the processor should idle, and for how much it should run. Since the amount of idle time is computed on a *per-core* basis, it is not yet discriminating which jobs are allowed to run faster and which are instead slowed down, but it is only given and advice to the scheduler on when to preempt the running task. Moreover, some classes of threads may not be preempted at all in this way, for example threads belonging to the kernel and realtime scheduling classes.

Two observations are in order here:

- the control action is a suggestion of ADAPTME to the scheduler, which means that the goal may not be achieved. It is by all means a "best-effort" policy. However, under normal conditions, i.e. when kernel threads and realtime threads do not consume too much CPU time, the policy, as we show in Chapter 6, achieve its goal. This is further discussed after introducing the second policy, since this behavior may be obtain by adjusting one of ADAPTME parameters.

- the control policy is run *asynchronously* with respect to the scheduling main loop, so that the quantum of time is not linked to any particular timeshare-class thread (i.e. the kind of thread that may be spawned by a user space program).

As it is described Chapter 5, it is implemented the control strategy in different parts of the scheduler, in order to cope with all the possible transitions of state of the processes.

4.3.2 Performance-aware policy

As already said, it was employed HRM as the monitoring infrastructure for getting informations about the current status of system's applications. Thanks to the heart rate metaphor, it is possible to control the amount of CPU time allowed to an application in order for it to reach its goals, compatibly with the other non-legacy applications' goals.

The controller described is capable of adjusting this amount of time by varying the priority of the application, as it will be thoroughly described in Chapter 5. Many different portions of the base scheduler's code were modified in order to cope with the various modalities in which the original module accounted for priority updating.

Two observations are in order here:

- the control action is a suggestion of ADAPTME to the scheduler, which means that the goal may not be achieved. It is by all means a "best-effort" policy. As it is thoroughly described in [3], where the author implements this policy on a commodity GNU/Linux operating system in a similar fashion, this policy can effectively favor those applications that are currently not yet reaching their goals. As we show in Chapter 6, the same kind of behavior is observed with ADAPTME, confirming the feasibility of this approach.

- the control policy is run *synchronously* with respect to the scheduling main loop, so that the priorities are modified in the same way as was done in the original scheduler, only with a different update equation (which is that of 4.5).

Chapter 5

Implementation

In this chapter it is described the relevant portions of the implementation of ADAPTME. In order to introduce the previously described self-awareness features to FreeBSD, Heart Rate Monitor (HRM) has been ported from Linux (both user space and kernel space) and its peculiarities are detailed in section 5.1. The 4.4BSD scheduler implementation is detailed in section 5.2, as a precondition for the subsequent discussion about the implementation details of ADAPTME. The implementation details required to introduce temperature- and performance-awareness are detailed in section 5.3.

5.1 FreeBSD Heart Rate Monitor porting

As it happens in the Linux version of HRM, FreeBSD's HRM porting consists of two partitions, the user-space one and the kernel-space one [82].

The user-space partition of HRM is targeted at both software and system developers, who are required to access its functionalities by means of a library, namely `libhrm`. This library exposes a compact C Application Programming Interface (API) that grants software and system developers the ability to instrument applications, providing a way to specify performance goals, signal progresses, retrieve applications performance measures and retrieve performance goals. The basic API functions are reported in Table 5.1.

In the following two subsections, the focus is set on the internals of the user

space and the kernel space partitions of HRM.

5.1.1 Heart Rate Monitor user space partition

HRM can be accessed by userspace applications via `libhrm`. For the sake of portability, `libhrm` interface is the same under both Linux and FreeBSD, even though minor semantics differences exist in the current version of the implementation.

The typical usage workflow is summarized in pseudo algorithm 1.

Pseudocode 1 HRM: application's perspective (producer and/or consumer)

```
init-attach: hrm_attach
init-goal: hrm_set_min_heart_rate and hrm_set_max_heart_rate
init-window: hrm_set_window_size
while (not done) do
  if (producer) then
    emit hartbeats: heatbeat or heartbeatN
  end if
  if (consumer) then
    read heart rates: hrm_get_window_heart_rate or
    hrm_get_global_rate
  end if
end while
detach from group: hrm_detach
```

Observe that the behavior of the application is dependent on the mode it has attached to the group; if it has attached in *producer* mode, the application will be able to issue calls to any function of `libhrm`, else it will not be able to invoke `heartbeat` and the various `hrm_set_*` functions.

In pseudo algorithm 1, the sample application attaches to a given numbered group (as defined in 2.4.1) and, during execution, emits heartbeats to signal some kind of progress, if it is a *producer*, and reads the current window or global heart rate if it is a *consumer*. On exit, the application detaches from the group.

The proposed API exposes two functions, namely `hrm_attach` and `hrm_detach`, the first to attach the issuing threads to the group identified by a Group IDentifier (GID) in consumer or producer mode, and the latter to detach the caller from the group it is assigned to, respectively.

Table 5.1: *libhrm* API

Function	Description
`hrm_attach(int gid, bool_t consumer)`	Attach the current task to the group identified by `gid`.
`hrm_detach()`[1,2]	Detach the current task from a given monitor instance
`hrm_set_min_heart_rate(uint32_t min_heart_rate)`[1,2,3]	Set the minimum heart rate in the user–defined performance goal
`hrm_set_max_heart_rate(uint32_t max_heart_rate)`[1,2,3]	Set the maximum heart rate in the user–defined performance goal
`hrm_set_window_size(size_t window_size)`[1,2,3]	Set the window size in the user–defined performance goal
`hrm_get_min_heart_rate()`[1,2]	Get the minimum of the target heart rate
`hrm_get_max_heart_rate()`[1,2]	Get the maximum of the target heart rate
`hrm_get_window_size()`[1]	Get the window size
`hrm_get_global_heart_rate()`[1,2]	Get the global heart rate
`hrm_get_window_heart_rate()`[1,2]	Get the window heart rate
`heartbeat(uint64_t n)`[1,2,3]	Emit n heartbeats

[1]Every function receive an additional parameter of type `hrm_t *` pointing to the underlying monitor data structure
[2]Every function return a value of type `int` containing either 0 or an error number
[3]Every task attached as a consumer is not allowed to call this function

Different applications may be concerned with either long- or short-term trends. For example, a process that has a periodic behavior may be interested both at the average heart rate reached during overall execution and short term heart rate over, for example, the last tens of seconds. Therefore, the API exposes two suitable functions, namely `hrm_get_global_rate`, to catch long-term trends by averaging the heart rate over the whole application's execution time, and `hrm_get_window_heart_rate`, to catch its short-term trends (i.e., 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 can be set through a call to a `libhrm` function, namely `hrm_set_window_size`.

Two additional functions are exposed to adjust performance goals. The first is `hrm_set_min_heart_rate` which allows to define the minimum desired heart rate. The second is `hrm_set_max_heart_rate`, which allows to define the maximum desired heart rate. Conversely, the two procedures `hrm_get_min_heart_rate` and `hrm_get_max_heart_rate` allow the user to retrieve the current goal.

The most important API function is `heartbeat`. Calls to this function are inserted within the hotspot of a program to signal progresses by incrementing the summation of heartbeats by a generic integer value.

The complete set of `libhrm` functions belonging to the API is reported in Table 5.1.

Due to the lack in FreeBSD of a well developed and stable `procfs` virtual filesystem mechanism such as that offered by Linux, mapping memory pages between kernel space and user space is implemented in a different way. While in Linux this exchange of data is accomplished by reading and writing files belonging to the `procfs` filesystem following a protocol known by both the kernel and `libhrm`, in FreeBSD we rely on the creation of virtual devices under `/dev` that are suitably read and written to exchange information with the kernel. This way, even though `libhrm` is implemented in different ways in FreeBSD and in Linux, the interface remains the same, thus achieving to some degree the goal of *portability*.

5.1.2 Heart Rate Monitor kernel space partition

Pseudocode 2 HRM: system's perspective

```
loop
  if (task attachment request) then
    add task to existing group/create group if not existing: hrm_add_task_to_group
  end if
  if (task emits heartbeat in userspace) then
    recompute heart rate, global and window, of the associated group
  end if
  if (task detachment request) then
    remove task from existing group/remove group if empty: hrm_delete_task_from_group
  end if
end loop
```

The pseudo algorithm 2 represents the usage of HRM, as seen from the point of view of the kernel. Whenever a task requires to be added to a group, if it exist, the performance counter of that group is attached to the address space of that task, otherwise the group and the statistics page are created and then mapped to that task's address memory. After that, the system responds to incoming heartbeats requests, i.e. it must add the requested amount of heartbeats to the performance counter of the corresponding group. Whenever tasks request detachment, the system takes care of the removal of the task from the corresponding group

and, in case it is empty, removes that group from the list of available groups.

The kernel-space partition of HRM consists of all the data structures and management routines for realizing the performance monitoring. Excluding kernel-specific types and data structures, the FreeBSD port of the kernel-space partition of HRM overlaps that of the Linux implementation.

The basic data structure behind HRM is a linked-list of groups, shown in Figure 5.1.

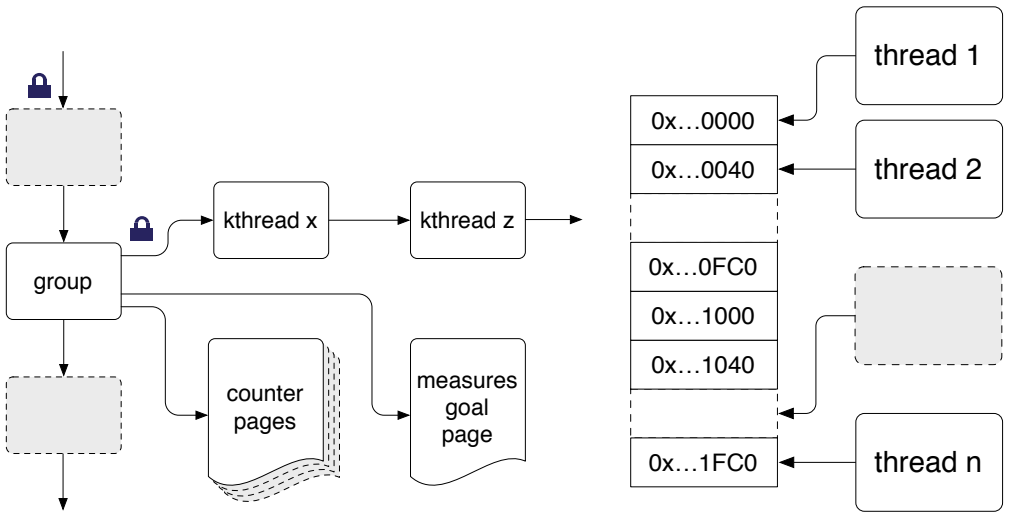

Figure 5.1: On the left, the linked-list of groups and the linked-list of producers and consumers (per group), set of pages to store per thread counters, and single page to store both the performance measures and the performance goal. On the right, the shared memory access pattern realized by threads belonging to the same group accessing their respective performance counters.

Since HRM data structures may be accessed in a multithreaded fashion, the linked-list of groups is protected by a global mutex, which is needed to guarantee consistency when modifying the linked-list during group addition or deletion.

One major difference with the Linux implementation of HRM data structures is the absence of a second list of consumer-only threads. This is because of it was not possible to conclude the implementation of this feature in time. From the user's point of view, what happens is that attaching to a group in consumer mode is simply not possible; users can attach in producer mode only, which allows them to exploit all of HRM exposed procedures. The difference between the two is purely performance-related: whenever an adaptation policy is traversing the

list of group's tasks, it may probably not be interested in accessing tasks which are consumers-only. Apart from this minor difference, the behavior of the system from the point of view of the user is fundamentally the same.

Each group allocates a set of pages to store per thread counters (i.e., default 1 page, but this may grow up to 16 pages) and a single page to store both the performance measures and the performance goal as depicted in Figure 5.1. These pages are shared across the kernel and the user address spaces via a shared memory mechanism; moreover, since HRM supports diverse parallelization models through the concept of group, pages may be shared across multiple user address spaces since each of the user thread may belong to a different process [82].

The right part of Figure 5.1 gives a more accurate view of the layout of the pages storing the counters; as the least significant portion of the addresses highlight, each counter is aligned to the size of the cache line. Cache line alignment results in a slightly less efficient use of the available memory; however, the claim is the performance improvements due to cache line alignment on multi and many-core processor and especially on multi-processor systems, which necessitate off-chip communication to maintain cache coherency, is such that more memory can be allocated, being an increasingly available resource in modern computing systems. The content of the pages devoted to store the counters is the most critical to HRM since it can be concurrently accessed at a high rate by both the kernel and user threads. Distribution avoid synchronization among user threads, while heavy weight synchronizations between kernel and userspace are avoided by adopting atomic operations; hence, a function call to heartbeat reduces to an atomic increment of a per thread counter. Due to cache line alignment, the number of counters is architecture dependent; the reference implementation of HRM allocates standard sized pages whose size is 4 kB, while the size of cache lines of x86 and x86-64 processors is 64 bytes, with such parameters, each page can contain up to 64 counters [82].

In addition, as the left part of Figure 5.1 shows, each group allocates a single page to be shared with the group to store both the performance measures and the performance goal; this is different with respect to what happens with

the counters, since they are distributed among the group's tasks. As reported in Section 2.4, HRM provides both a global heart rate (i.e., longterm performance measure) and a window heart rate (i.e., short-term performance measure) computed according to Equations 5.1 and 5.2 in which g indicates the group, t is the current time stamp, t_0 is the group creation time stamp, and t_w is the time stamp at which window started [82].

$$ghr_g(t) = \sum_i \frac{cnt_i(t)}{t - t_0} \tag{5.1}$$

$$whr_g(t) = \sum_i \frac{cnt_i(t) - cnt_i(t_w)}{t - t_w} \tag{5.2}$$

The performance measures are asynchronously updated by the kernel in the context of a HighResolution (HR) timer after acquiring the members read-write lock in read mode; the adoption of asynchronous updates for performance measures avoids boundary crosses to retrieve the current time stamp. The period of the HR timer can be tuned through a kernel compile time parameter. The performance goal is made up of a lower and an upper bound defining a heart rate range; moreover, the performance goal contains also the window size to compute the window heart rate [82].

HRM implements a number of virtual device files to provide all the necessary entry points to attach (detach) threads to (from) a group and `mmap` pages storing per thread counters and both the performance measures and the performance goal.

Some other portions of FreeBSD have been modified in order to flawlessly integrate the monitoring system with the rest of the Operating System (OS):

- In order to allow applications to interface to kernel space data, virtual devices are instantiated at run time. It was not possible to implement or reutilize code from Linux, since it relies on the presence of a virtual filesystem (`procfs`) to exchange informations to/from kernel space via `procfs`, a feature not available in FreeBSD 7.2.

- the `struct thread` data structure, which is the fundamental data struc-

ture describing threads in FreeBSD, has been modified to contain a new field holding all HRM-related informations, namely `struct hrm_producer`.

- a virtual device (`/dev/hrm`) has been implemented to easily report on the overall current activity of all HRM groups: it is possible to open this device in read only mode to obtain a formatted string containing informations about target, global and window heart rate, goals and tasks of all available tasks of all available groups.

Listing 5.1 is an excerpt of code from `hrm.h`. Line 1 declares and instantiates the list of `hrm_group` mentioned above. At line 24 begins the declaration of what a group is: it contains a (pointer to) performance counters, goals of the tasks, a history of previous heart rates for computing the window heart rate value, the list of producer tasks (consumers are going to be implemented in the next iteration of ADAPTME). The producers are described by the data structure `hrm_producer`: there is a pointer to the performance counter, the pointer to the window and global heart rate and the goal of the task. Finally, a pointer to the group which the thread belongs to.

```
1  LIST_HEAD(hrm_groups_list, hrm_group);
2
3  extern struct hrm_groups_list hrm_groups;
4  extern struct mtx hrm_groups_lock;
5
6  struct hrm_memory_map;
7  struct hrm_memory_map {
8      lwpid_t pid;
9      unsigned long user_address;
10     int references;
11     LIST_ENTRY(hrm_memory_map) link;
12 };
13
14 struct hrm_memory {
15     unsigned long kernel_address;
16     size_t size;
17     LIST_HEAD(hrm_memory_map_list, hrm_memory_map) maps;
18 };
19
20 struct hrm_group;
21 struct hrm_group {
22     int gid;
23     struct hrm_memory counters;
24     struct hrm_memory measures_goal;
25     struct {
26         int window_begin;
```

```
27          int window_end;
28
29          struct {
30              uint64_t counter;
31              struct timespec elapsed_time;
32          } window[HRM_WINDOW_SIZE];
33
34          uint64_t history;
35      } history;
36
37      struct callout timer;
38      struct timespec elapsed_time;
39      struct timespec timestamp;
40
41      LIST_HEAD(hrm_producers_list, hrm_producer) producers;
42      //LIST_HEAD(hrm_consumers_list, hrm_consumer) consumers; -- HRM v2.0, not yet implemented
43      struct mtx members_lock;
44
45      LIST_ENTRY(hrm_group) link;
46  };
47
48  struct hrm_counter {
49      lwpid_t tid;
50      int used;
51      uint64_t counter;
52  };
53
54  struct hrm_measures {
55      uint32_t global_heart_rate;
56      uint32_t window_heart_rate;
57  };
58
59  struct hrm_goal {
60      uint32_t min_heart_rate;
61      uint32_t max_heart_rate;
62      size_t window_size;
63  };
64
65  struct hrm_producer;
66  struct hrm_producer {
67      int counter_index;
68
69      struct hrm_counter *counter;
70      struct hrm_measures *measures;
71      struct hrm_goal *goal;
72      struct hrm_group *group;
73
74      unsigned long counter_user_address;
75      unsigned long measures_user_address;
76      unsigned long goal_user_address;
77
78      LIST_ENTRY(hrm_producer) link;
79  };
```

Listing 5.1: HRM relevant data structures

5.2 4.4BSD scheduler

The basic notions regarding the 4.4BSD scheduler were already given in Section 2.1.2. Here follows a deeper overview of the implementation of the scheduler.

5.2.1 Multilevel feedback Run Queues

The most important data structure employed for the scheduling process is the *multilevel feedback run queue*. This is a queueing scheme in which requests are partitioned into multiple prioritized subqueues, with requests moving between subqueues based on dynamically varying criteria [35].

The run queues contain all the runnable threads in main memory except the currently running thread. They are organized as a doubly linked list of thread structures. The head of each run queue is kept in an array of fixed (known at compile-time) dimensions. The number of queues used to hold the collection of all runnable threads in the system affects the cost of managing the queues. If only a single (ordered) queue is maintained, then selecting the next runnable thread becomes simple but other operations become expensive. Using 256 different queues can significantly increase the cost of identifying the next thread to run. The system uses 64 run queues, selecting a run queue for a thread by dividing the thread's priority by 4. To save time, the threads on each queue are not further sorted by their priorities [35].

Associated with this array is a bit vector, `rq_status`, that is used in identifying the nonempty run queues, so as to speed up the process of task traversal. Two routines, `runq_add()` and `runq_remove()`, are used to place a thread at the tail of a run queue, and to take a thread off the head of a run queue. In order to choose the next thread to run, the scheduling algorithm calls the `runq_choose()` routine. The `runq_choose()` routine operates as follows [35]:

- Ensure that our caller acquired the `sched_lock`
- Locate a nonempty run queue by finding the location of the first nonzero bit in the `rq_status` bit vector. If `rq_status` is zero, there are no threads

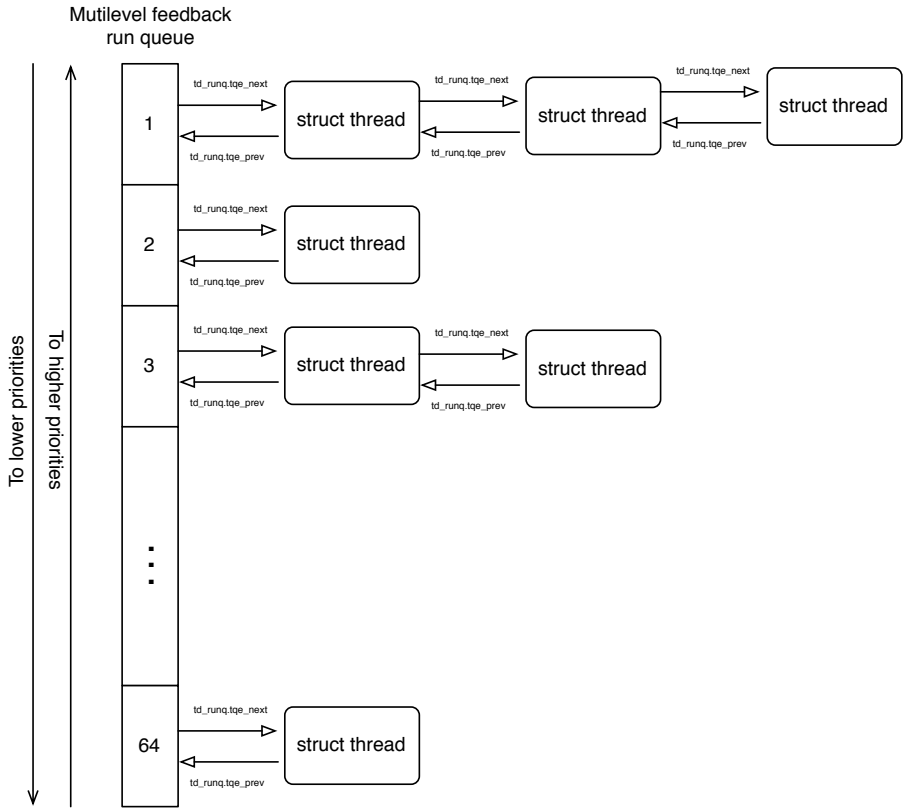

Figure 5.2: Multilevel feedback run queues: tasks are assigned a level in the queue and are double linked for round robin scheduling purposes.

to run (in this case select the idle loop thread)

- Given a nonempty run queue, remove the first thread on the queue.
- If this run queue is now empty as a result of removing the thread, reset the appropriate bit in `rq_status`
- Return the selected thread and release the lock.

5.2.2 Computation of threads' priority

After describing how the scheduler physically keeps threads ordered on the basis of their priority, it is discussed how the scheduler recomputes priorities. Different functions are responsible for priority calculations used in the short-term scheduling algorithm.

The scheduler needs to carry out the following tasks:

- periodic recomputation of threads' priority, based upon the effective running and sleeping time
- periodic preemption of tasks to allow others at same priority to run in a round robin fashion
- statistics bookkeeping (system load, effective amount of quantum used by process etc...)
- preemption whenever a task with higher priority is put in the ready queue

For the first tasks, two routines, `schedcpu()` and `roundrobin()`, run periodically. `schedcpu()` is a high priority kthread that runs at a frequency of 1Hz; the main loop of the thread locks processes and threads belonging to them one at a time, and recomputes their priorities using Equation 2.1. Moreover, it updates the value of `kg_slptime` for threads blocked by a call to `sleep()` [35].

The `roundrobin()` routine carries out the second task: it runs 10 times per second and causes the system to choose the next thread to run in the highest-priority (nonempty) run queue in a round-robin fashion, which allows each thread a 100-millisecond time quantum [35].

The third tasks comprises Central Processing Unit (CPU) usage estimates, which are updated in the system clock-processing module, `hardclock()`, executing 100 times per second. Each time that a thread accumulates four ticks in its CPU usage estimate, `kg_estcpu`, the system recalculates the priority of the thread. This recalculation uses Equation 2.1 and is done by the `resetpriority()` routine [35].

In addition to issuing the call from `hardclock()`, each time `setrunnable()` places a thread on a run queue, it also calls `resetpriority()` to recompute the thread's scheduling priority. This call from `wakeup()` to `setrunnable()` operates on a thread other than the currently running thread. So `setrunnable()` invokes `updatepri()` to recalculate the CPU usage estimate according to equation 2.2 before calling `resetpriority()`. The relationship between these functions is shown in Figure 5.3.

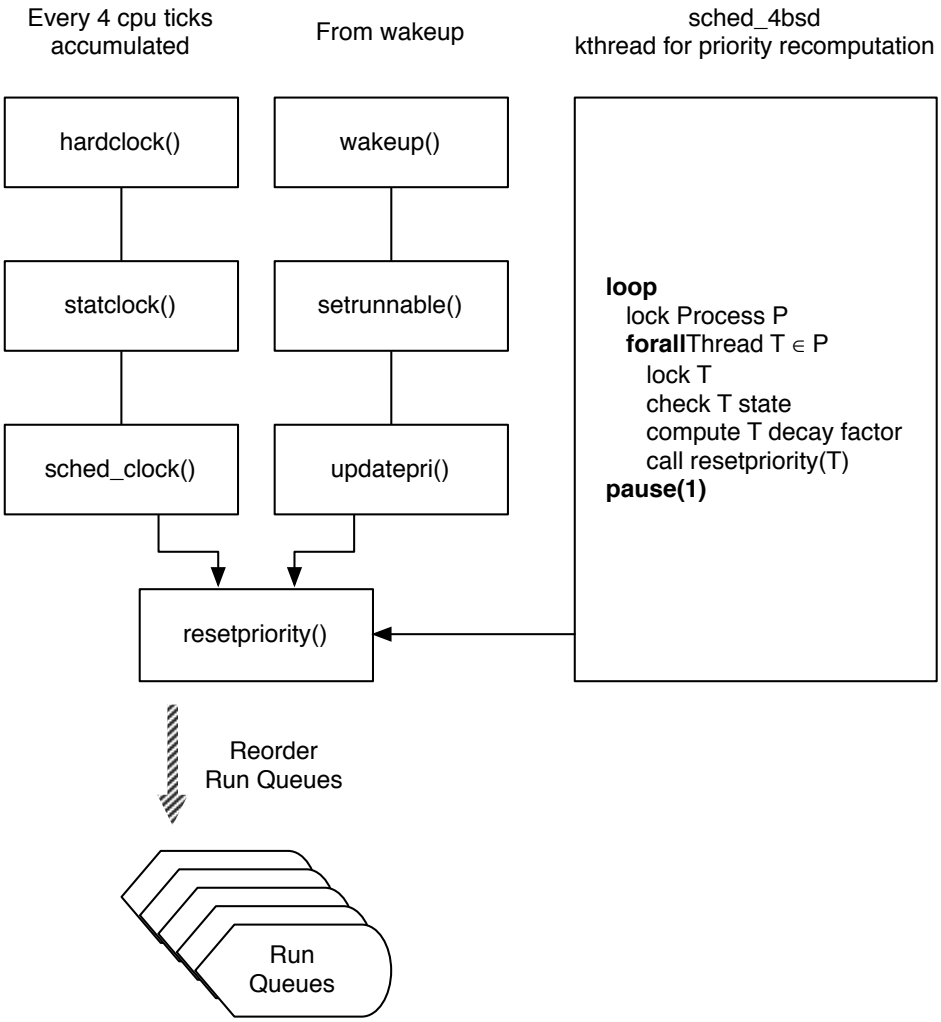

Figure 5.3: The relationship between some relevant scheduler functions

5.3 ADAPTME implementation

ADAPTME was implemented in the FreeBSD 7.2 kernel, modifying the operations of the 4.4BSD scheduler to make a fair comparison with Dimetrodon [22], which was implemented on the same release of that OS. The 4.4BSD scheduler is augmented with the thermal-aware policy and the performance-aware policy.

5.3.1 Performance-Aware Policy

Within the 4.4BSD scheduler, all threads that are runnable are assigned a scheduling priority that determines in which run queue they are placed. Since FreeBSD 6, run queues in Simultaneous Multi Processor (SMP) systems are assigned one per core. In selecting the new thread to run, the scheduling infrastructure scans the run queues from the highest to the lowest priority and chooses the first thread on the first nonempty run queue. To speed up the search, an helper bit array called `rq_status` is present, where each element is a 0 in case the corresponding run queue is empty and 1 otherwise. Multiple threads on the same run queue are managed in a round robin fashion and are assigned a static quantum of time, which is decided at compile time. The 4.4BSD scheduler is based on the multilevel feedback queues infrastructure already introduced in subection 5.2.1; threads migrate among run queues according to their changing scheduling priority. Higher priority threads preempt lower priority threads whenever they are added on a run queue. Since it was being extended an existing scheduling infrastructure it was put a lot of effort into preserving all the desirable properties coming with the 4.4BSD scheduler (e.g., non-starvation, priority decay). Keeping this in mind, it was decided, for example, to implement the performance aware policy as a mechanism that variates the priority of threads, rather than the length of the quantum, so as to preserve the semantic of this OS element.

The performance-aware policy is an extension of the scheduling infrastructure acting in a decoupled fashion. Threads priorities, which are updated at a constant rate, are adjusted using an additive term $\texttt{priority}_{i,j}$ for each thread j of application i, where the application is a non-legacy one (i.e. instrumented with

HRM). This operation migrates threads from the current run queue to another one, advantaging or disadvantaging threads according to their measured performance and performance goals. If the application's heart rate is higher than that desired, the OS will just lower its priority, so as to favor the execution of other tasks; on the contrary, in case the application's heart rate is lower that the desired one, the scheduler will try its best to increase the priority of that process and favor its execution. This statement underlines the "best-effort" nature of this mechanism: in fact, if multiple applications are not yet meeting their goals, it is possible that they will never do so since not enough CPU time may be allowed to any of them to satisfy their requirement.

In the 4.4BSD scheduler, the priority of a thread indicates also which scheduling class it belongs to. There exist five scheduling classes: one for the bottom-half kernel threads, one for the top-half kernel threads, one for the real-time user threads, one for the time sharing user threads, and the last one for the idle threads. Since the behavior of the scheduling infrastructure varies with respect to the scheduling class the running thread belongs to, it is necessary to avoid scheduling class changes. Such changes can happen whenever the performance-aware policy, which is designed to work on time sharing user threads, adjusts priorities. In our solution, proper controls on the priority values avoid class variations. Each thread is further marked with a *force idle* flag if it is over performing or *prevent idle* flag if it is under performing in accordance with the output of the control-theoretical system, allowing for a stricter control over performance to take place. Notice, again, how this may imply that some tasks are never idled by the thermal policy, thus contributing to heating the system. Again, this is a "best-effort" approach that, as it will be seen in Chapter 6, under normal conditions works well enough.

Pseudo-code for priority update policy

Pseudocode 3 ADAPTME: performance-aware thread priority modification pseudo-code

```
loop
  if (sched_perf is enabled) then
    for all (p: process P) do
      for all (t: thread ∈ p) do
        compute delta performance/goal of t
        if (over maximum desired heart rate) then
          set t.TDF_FORCEIDLE to suggest to idle thread t
        end if
        if (under minimum desired heart rate) then
          set t.TDF_PREVENTIDLE to suggest not to idle thread t
        end if
        if (100th run the update priority thread) then
          update μ estimation
        end if
        compute delta performance as in eq. 4.4
        assign Δp_t to field hrm_delta_prio in struct thread, to be read
        during actual priority update
      end for
    end for
  end if
end loop
```

5.3.2 Thermal-Aware Policy

The thermal-aware policy acts in coordination with the 4.4BSD scheduler. When the scheduler chooses the next thread to run, it decides whether to run it or to run the idle thread preempting the selected thread. The idle thread is scheduled if the output of the control-theoretical system says to idle and the thread that is going to be preempted is not system critical nor it is marked with the *prevent idle* flag. The idle thread is also scheduled if the thread that is going to be preempted is marked with the *force idle* flag. A thread is considered system critical if it is a bottom-half kernel thread (e.g., interrupts) or a top-half kernel threads (common kernel threads such as that of priorities recomputation); moreover, since system critical kernel threads do not belong to non-legacy applications, they cannot be marked with either *force idle* or *prevent idle*.

The temperature-aware policy is a high priority kernel thread that realizes the control loop explained in subsection 4.3.1. This thread sets an appropriate field in the `struct thread` data structure of every thread which indicates the amount of effective execution time that will be spent by the thread the next time it is scheduled. After reading the temperature of all cores by accessing the appropriate processor register, it stores the value in an array that will be later used by the rest of the scheduler to take its decisions.

Different points in code are updated to cope with the thermal-aware scheduling mechanism. The relevant ones are reported here:

- in `sched_4.bsd@sched_clock`: if the thread has not yet used up its full quantum of time, it may be that it is anyway descheduled and the idle thread scheduled instead, depending on the thermal conditions of the system and the applied control.
- in `sched_4.bsd@sched_fork_thread`: the child of current thread inherits the father's thermal flags
- in `sched_4.bsd@sched_choose`: here the scheduler chooses the next thread to run; in case we are out of temperature target and the current thread may be idled, we return the idle thread as the next thread to run with a probability that is computed by the Control Theory (CT) based mechanism. This is the core of the thermal-aware policy.

Pseudo-code for temperature mitigation policy

Pseudocode 4 ADAPTME: thermal-aware policy pseudo-code

loop
read temperature of all cores
compute every 100 ms the amount of time that thread t has to spend idling
store value in `struct thread` data structure
during `sched_clock` and `sched_choose` decide whether to continue executing thread's code or idle thread
end loop

Chapter 6

Results

After describing the motivation, the working methodology and the implementation of ADAPTME, a description of the experiments that were set up to validate the thesis' work is given in this Chapter. After introducing the benchmark suite that was employed for the tests in Section 6.1, a description of the experimental settings for this work and Dimetrodon is given in Section 6.2. In order to better understand the rationale behind the experiments, the workloads are characterized in subsection Section 6.1.1. Finally, the results of the experiments are illustrated and commented in Section 6.3.

6.1 Benchmarking in a multicore environment: PARSEC

In choosing the benchmark that most suitably fits our needs, we identified the following requirements:

- As already explained in Section 3.2, it is required to implement a mechanism that allows applications to signal their progress and state their goals. Adding this capability to a software is called "instrumentation". Being able to instrument the benchmark is a major constraint, so a Free, Libre or Open Source Software (FLOSS) benchmark is required.
- In our system the focus is put on systems under heavy load conditions, where resources contention is particularly high. Under these conditions

many phenomena arise that may impact on performance and temperature (for example: added off-chip communication, uneven thermal maps etc...). A proper way to take into account this from a benchmark's point of view is to give users the ability to explicitly specify the amount of parallelism exposed by applications (for example: by requiring the user to state the number of threads of the test application).

- Since the focus is not put onto a particular workload's characteristic (like: lock contention, amount of Input/Output (I/O), particular distribution of assembly instructions etc...), the requirement is that our benchmark be as diverse and representative as possible.

Among the available FLOSS benchmarks, it was decided to use Princeton Application Repository for Shared-Memory Computers (PARSEC).

PARSEC is a state of art benchmark suite for studies of Chip Multi Processors (CMPs). Previous available benchmarks for multiprocessors have focused on high-performance computing applications and used a limited number of synchronization methods. PARSEC includes emerging applications in recognition, mining and synthesis (RMS) as well as systems applications which mimic large-scale multi-threaded commercial programs [4].

As the authors state in their reference work, [4], this benchmarking suite is superior to previous works (for example: SPLASH-2 [84], SPEC CPU2006 [85] and OMP2001) for the following reasons:

- All the applications belonging to the PARSEC suite have been parallelized, meaning that all of them may be executed using a given amount of threads to execute their job

- The PARSEC benchmark suite is not skewed towards HPC programs, which are abundant but represent only a niche. It focuses on emerging workloads. The algorithms these programs implement are usually considered useful, but their computational demands are prohibitively high on contemporary platforms. As more powerful processors become available in the near future, they are likely to proliferate rapidly [4].

- The workloads are diverse and were chosen from many different areas such as computer vision, media processing, computational finance, enterprise servers and animation physic [4]. In short, they are both representative and diverse.

6.1.1 Available PARSEC workloads

To date, the latest publicly available version of PARSEC is 2.1. The experimental results are all based on the workloads and data sets available in this version of the suite.

PARSEC comprises 13 applications and a manager to run tests according to user's specifications. Each of those 13 applications has its own datasets: there are 4 of these datasets per application, differing only for their dimensions (and total execution run time). These datasets allow for coarser and finer benchmarking, at the price of reduced and increased execution times, respectively.

In Table 6.1 we report the qualitative summary of the inherent characteristics of PARSEC benchmarks [4].

Table 6.1: Qualitative comparison between PARSEC workloads [4]

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high
raytrace	Media Processing	pipeline	fine	medium	medium	low

6.2 Settings

Now the focus is put on the experiments settings, describing the conditions under which the system was validated. In particular, since the claim in the Sum-

mary is that the implemented system performs better than the most similar work that it was found in the state of art, Dimetrodon [22], while introducing additional performance-aware capabilities, it is described how we managed to compare our work to it.

As it was described throughout Chapter 5, it was implemented ADAPTME in a commodity operating system, namely FreeBSD 7.2. Our real-world parallel workloads are those found in PARSEC 2.1. Before instrumenting the applications, a port of the suite to FreeBSD was required. Not all applications were designed to be portable, and others are parallelized by means of OpenMP and not pthreads (which is our reference threading model).For these reasons, it could be possible to instrument only a portion of the benchmark, namely blackscholes, fluidanimate, swaptions, ferret and x264, since in the first place. As it can be seen from Table 6.1, these five applications are representative for both data-parallel and pipelined parallelization models, fine, medium and coarse parallelization granularity, all dimensions of the working set, both low and high data usage sharing patterns and low, medium and high data usage exchange amounts. For this reason, the chosen subset of benchmarks' applications is nonetheless a representative one.

The tests of ADAPTME were done on an entry-level mid-tower server equipped with a single Intel Core i7-990X six-core processor running at 3.46 GHz with a nominal maximum Thermal Design Power (TDP) of 130 W, 6 GB of DDR3-1066 non-ECC RAM, and a 500 GB 7200 RPM SATA2 hard disk. Advanced features such as Intel Hyper-Threading Technology and Intel Turbo Boost Technology were disabled while Enhanced Intel SpeedStep Technology was enabled to allow the processor to enter and exit low-power modes.

To fairly compare the work with Dimetrodon [22], their latest version of their patch for FreeBSD 7.2 (which can be found here [86]) was obtained and the system was patched and compiled with all optimizations on. As Dimetrodon requires to set a number of parameters for it to work properly, they were empirically determined in order to obtain the best tradeoff between temperature reduction and runtime execution slowdown for each workload used for testing ADAPTME. When it had been possible, it was tried to make Dimetrodon obtain

a temperature reduction compatible with that of ADAPTME's, so as to favor this objective when choosing the right set of parameters.

Moreover, ADAPTME's thermal-policy was configured so that it would run as high-priority kernel thread. Temperature sampling period is 100 ms.

It was evaluated the thermal-aware policy of ADAPTME under the subset of the applications of PARSEC introduced at the beginning of this section. Each run consisted of 5 consecutive executions of the same application registering the machine thermal profile. The same experiments were repeated with FreeBSD/4.4BSD, Dimetrodon and ADAPTME. The thermal-aware policy of ADAPTME was configured with a target temperature close to the average temperature measured when Dimetrodon was executed.

It was evaluated also the performance aware policy of ADPTME by signaling inside the instrumented applications of the benchmark both a target and a current heart rate, in line with what described in Section 2.4 when discussing Heart Rate Monitor (HRM). In order to test whether the policy works as expected, we first run the benchmark application alone, and record the peak heart rate $\mathrm{hr}_{\mathrm{peak}}$. Then, we launched 4 instances of the same application; of course, the measured heart rate of each instance is $\mathrm{hr}_{\mathrm{peak}}/4$. We than specified two different goals: for the first group of three applications, we set a goal that is *lower* than $\mathrm{hr}_{\mathrm{peak}}/4$, while we specified a goal which is *higher* than $\mathrm{hr}_{\mathrm{peak}}/4$ for the remaining one. In this way, we are substantially favoring the execution time of one of the instances at the expenses of the other. What we want to see is that in this case the performance aware policy and the temperature aware policy work in pair giving higher priority to the application for which a higher goal is specified while meeting the temperature constraint.

Advanced features such as Intel Hyper-Threading Technology and Intel Turbo Boost Technology were disabled while Enhanced Intel SpeedStep Technology was enabled to allow the processor to enter and exit low-power modes.

6.3 Experimental Results

They are now described the 6 experiments that have been carried out in this work. The first five directly compare with Dimetrodon and the focus is put on the thermal aspects of ADAPTME, for a fairer comparison. The last experiment is focus on the capability to simultaneous control both performance and thermal aspects of the runtime system.

6.3.1 x264

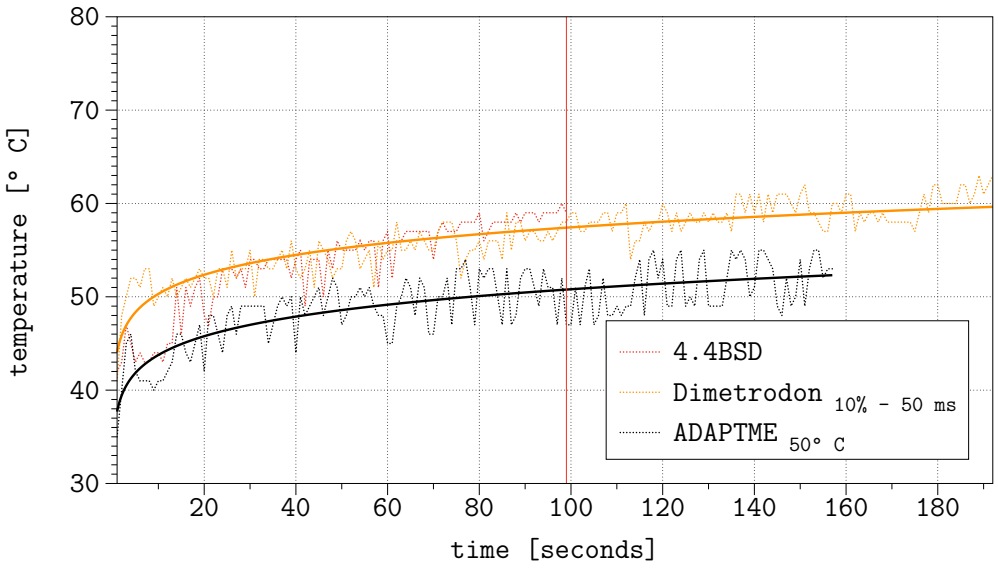

Figure 6.1: x264 run results. System's temperature of both Dimetrodon and FreeBSD/4.4BSD is consistently higher than that reached by ADAPTME. Notice how under ADAPTME the execution time of the workload is also reduced with respect to Dimetrodon.

The first experiment is done by running an instance of x264 under FreeBSD 7.2/4.4BSD, Dimetrodon and ADAPTME. This application is an H.264/AVC (Advanced Video Coding) video encoder whose flexibility, wide range of applications and ubiquity in next generation video systems are the reasons for its inclusion in the PARSEC benchmark suite. This workload has particular interdependences between data and a pipelined parallelization model that makes it partly I/O-bound and partly Central Processing Unit (CPU)-bound

In the first case, the scheduler is compiled into the system with the default

values for all the parameters, and the application benchmark is the only relevant application run during the experiment. 6 threads were assigned to this application for the test.

Dimetrodon was empirically configured so that the parameters would yield a an average temperature near that of ADAPTME's set point, without impacting too much on execution time. In our experiments, Dimetrodon cannot achieve both comparable execution time and average temperature, so it was decided to consider as a viable choice for its parameters a set of values by which the temperature was neither not too higher than ADAPTME's nor execution time too longer than ADAPTME's.

ADAPTME was configured so that the set point of the thermal-aware policy is 50° C. Being the only running application and being interested in focusing only on the thermal aspects of the work, a goal is set for the application that would be met anyway (min heart rate: 1 heartbeat per second; max: 10^8 heartbeats per second) so that the performance-aware policy is actually ineffective.

In Figure 6.1 the obtained results are shown. As expected, FreeBSD/4.4BSD has the lowest execution time at the expense of temperature, which reaches 60°C . This is compatible with the notion of "*run-to-idle*" scheduler. Dimetrodon, in this case, not only does not achieve its stated goal of lowering the temperature, but since the system randomly injects idle cycles during execution, the system experiences an overall slowdown by a factor of 1.8. On the other hand, ADAPTME successfully lowers the average and peak temperature by 8° C, oscillating around its goal, while slowing down the application by a smaller factor of 1.55.

6.3.2 `ferret`

The second experiment is carried out by running an instance of ferret under FreeBSD 7.2/4.4BSD, Dimetrodon and ADAPTME. This application is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data such as audio, images, video, 3D shapes and so on. The reason for the inclusion in the benchmark is that it represents emerging next generation desktop and Internet search engines for non-text document data types [4]. This applica-

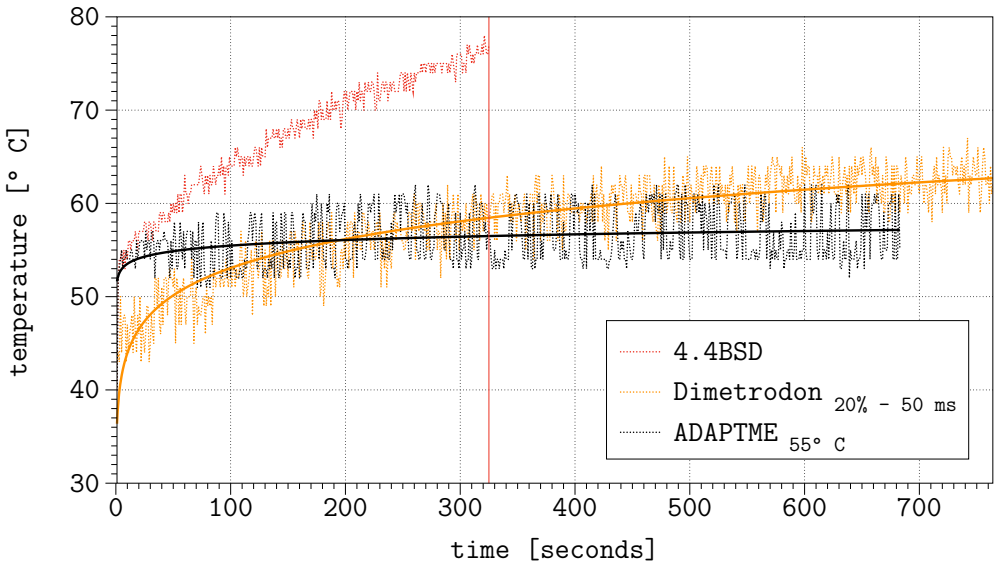

Figure 6.2: ferret run results. Being a CPU intensive workload, the temperature reached by FreeBSD/4.4BSD is high. Both Dimetrodon and ADAPTME consistently reduce temperature during the execution of the system, even though the average temperature is lower for ADAPTME. Execution time is still reduced in ADAPTME than it is in Dimetrodon. The vertical red line indicates the time when FreeBSD/4.4BSD finishes executing the benchmark application.

tion has a low amount of data interdependency between threads and reduced amount of I/O, so it can be considered fully CPU-bound.

In the first case, the scheduler is compiled again into the system with the default values for all the parameters, and the application benchmark is the only relevant application run during the experiment. 6 threads were assigned to this application for the test.

Dimetrodon was empirically configured so that the parameters would yield a an average temperature near that of ADAPTME's set point, without impacting too much on execution time. It was decided again to consider as a viable choice for its parameters a set of values by which the temperature was neither not too higher than ADAPTME's nor execution time too longer than ADAPTME's.

ADAPTME was configured so that the set point of the thermal-aware policy is 55° C. Being the only running application and being interested in focusing only on the thermal aspects of the work, it is set a goal for the application that would be met anyway (min heart rate: 1 heartbeat per second; max: 10^8 heartbeats per second) so that the performance-aware policy is actually ineffective.

In Figure 6.2 the obtained results are shown. As expected, FreeBSD/4.4BSD has the lowest execution time at the expense of temperature, which reaches 78°C. This is again compatible with the notion of "*run-to-idle*" scheduler. Dimetrodon, in this case, achieves its stated goal of lowering the temperature (by 15°C), but since the system randomly injects idle cycles during execution, the system experiences an overall slowdown by a factor of 2.38. On the other hand, ADAPTME successfully lowers the average and peak temperature by 22°C, oscillating around its goal, while slowing down the application by a smaller factor of 2.07.

6.3.3 `blackscholes`

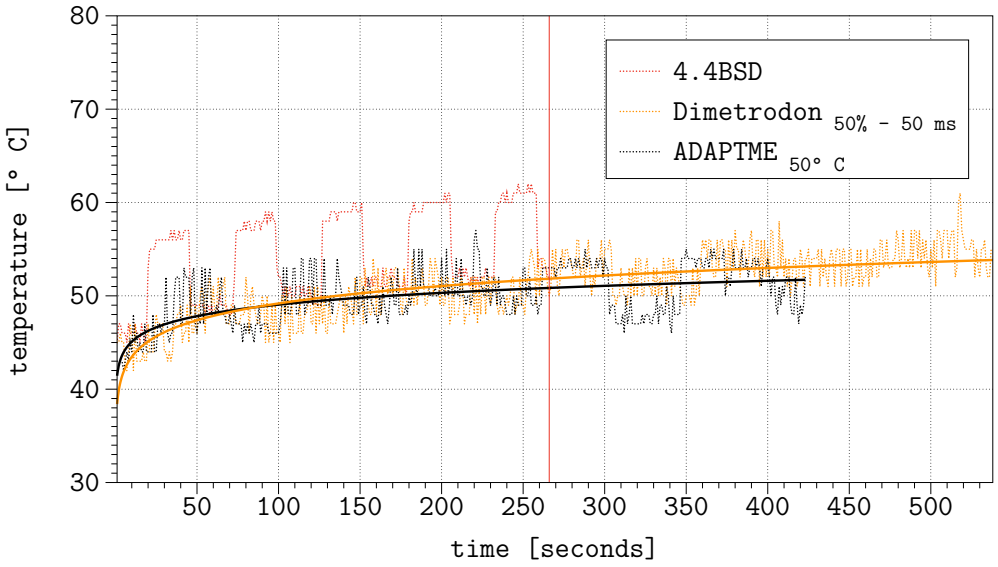

Figure 6.3: blackscholes run results. Again, the temperature reached by FreeBSD/4.4BSD is consistently higher than of both ADAPTME and Dimetrodon. In this case, we observe a reduction in execution time of ADAPTME with respect to Dimetrodon, which effectively shows how the performance aware policy can favor the execution of some tasks at the expense of others. The vertical red line indicates the time when FreeBSD/4.4BSD finishes executing the benchmark application.

The third experiment is done by running an instance of blackscholes under FreeBSD 7.2/4.4BSD, Dimetrodon and ADAPTME. This application is an Intel Recognition, Mining and Synthesis (RMS) benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). This application has a low amount of data interdepen-

dency between threads and reduced amount of I/O, so it can be considered fully CPU-bound.

In the first case, the scheduler is compiled again into the system with the default values for all the parameters, and the application benchmark is the only relevant application run during the experiment. 6 threads were assigned to this application for the test.

Dimetrodon was empirically configured so that the parameters would yield a an average temperature near that of ADAPTME's set point, without impacting too much on execution time. It was decided again to consider as a viable choice for its parameters a set of values by which the temperature was neither not too higher than ADAPTME's nor execution time too longer than ADAPTME's.

ADAPTME was configured so that the set point of the thermal-aware policy is 50° C. Being the only running application and being interested in focusing only on the thermal aspects of the work, it is set a goal for the application that would be met anyway (min heart rate: 1 heartbeat per second; max: 10^8 heartbeats per second) so that the performance-aware policy is actually ineffective.

In Figure 6.3 the obtained results are shown. As expected, FreeBSD/4.4BSD has the lowest execution time at the expense of temperature, which reaches 61°C. Observe how the subsequent runs in FreeBSD/4.4BSD causes the system to quickly heat and quickly cool down. During the spikes the system activates cooling mechanisms such as fans or air conditioners to lower the temperature and avoid electronics break down, with consequent waste of power. This is again compatible with the notion of "*run-to-idle*" scheduler. Dimetrodon, in this case, achieves its stated goal of lowering the temperature (by 7°C), but since the system randomly injects idle cycles during execution, the system experiences an overall slowdown by a factor of 2.03. On the other hand, ADAPTME successfully lowers the average and peak temperature by 10°C, oscillating around its goal, while slowing down the application by a smaller factor of 1.58.

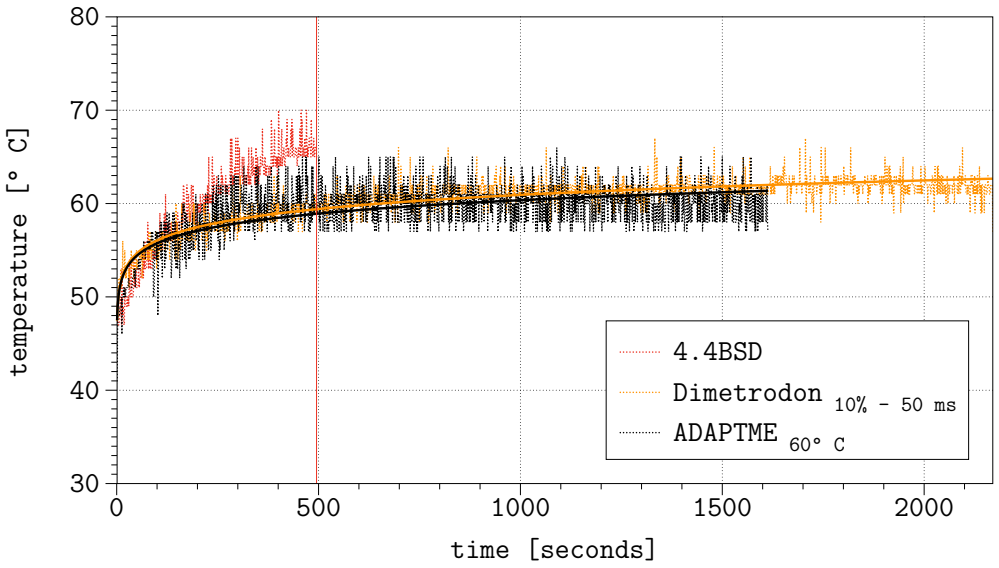

Figure 6.4: fluidanimate run results. Again, the temperature reached by FreeBSD/4.4BSD is consistently higher than of both ADAPTME and Dimetrodon. In this case, it is observed a reduction in execution time of ADAPTME with respect to Dimetrodon, which effectively shows how the performance aware policy can favor the execution of some tasks at the expense of others. The execution time overhead imposed by both ADAPTME and Dimetrodon is not negligible, being in the order of 3x – 5x. The vertical red line indicates the time when FreeBSD/4.4BSD finishes executing the benchmark application.

6.3.4 `fluidanimate`

The fourth experiment is carried out by running an instance of fluidanimate under FreeBSD 7.2/4.4BSD, Dimetrodon and ADAPTME. This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. This application has a low amount of data interdependency between threads and reduced amount of I/O, so it can be considered fully CPU-bound.

In the first case, the scheduler is compiled again into the system with the default values for all the parameters, and the application benchmark is the only relevant application run during the experiment. 8 threads were assigned to this application for the test.

Dimetrodon was empirically configured this time so that the parameters would yield the same temperature of ADAPTME's set point, without taking into account execution time overhead.

ADAPTME was configured so that the set point of the thermal-aware policy is 60° C. Being the only running application and being interested in focusing only on the thermal aspects of the work, a goal is set for the application that would be met anyway (min heart rate: 1 heartbeat per second; max: 10^8 heartbeats per second) so that the performance-aware policy is actually ineffective.

In Figure 6.4 the obtained results are shown. As expected, FreeBSD/4.4BSD has the lowest execution time at the expense of temperature, which reaches 70°C. Observe how the subsequent runs in FreeBSD/4.4BSD causes the system to quickly heat and quickly cool down. During this spike the system activates cooling mechanisms such as fans to lower the temperature and avoid electronics break down, with consequent waste of power due to increase in fan speed. This is again compatible with the notion of "*run-to-idle*" scheduler. Dimetrodon and ADAPTME achieve their stated goal of lowering the temperature (by 8°C), but since in Dimetrodon the system randomly injects idle cycles during execution, that system experiences an overall slowdown by a factor of 4.34. On the other hand, ADAPTME slows down the application by a smaller factor of 1.35.

6.3.5 `swaptions`

The fifth and last thermal oriented experiment is carried out by running an instance of swaptions under FreeBSD 7.2/4.4BSD, Dimetrodon and ADAPTME. The swaptions application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. This application has an average amount of data interdependency between threads and I/O, so it can be considered partly CPU-bound and partly I/O-bound.

In the first case, the scheduler is compiled again into the system with the default values for all the parameters, and the application benchmark is the only relevant application run during the experiment. 6 threads were assigned to this application for the test. Moreover, the parameters of this application were set so that a single run would last long enough to make the warm up and wind down of the processor's temperature negligible with respect to the rest of the execution time.

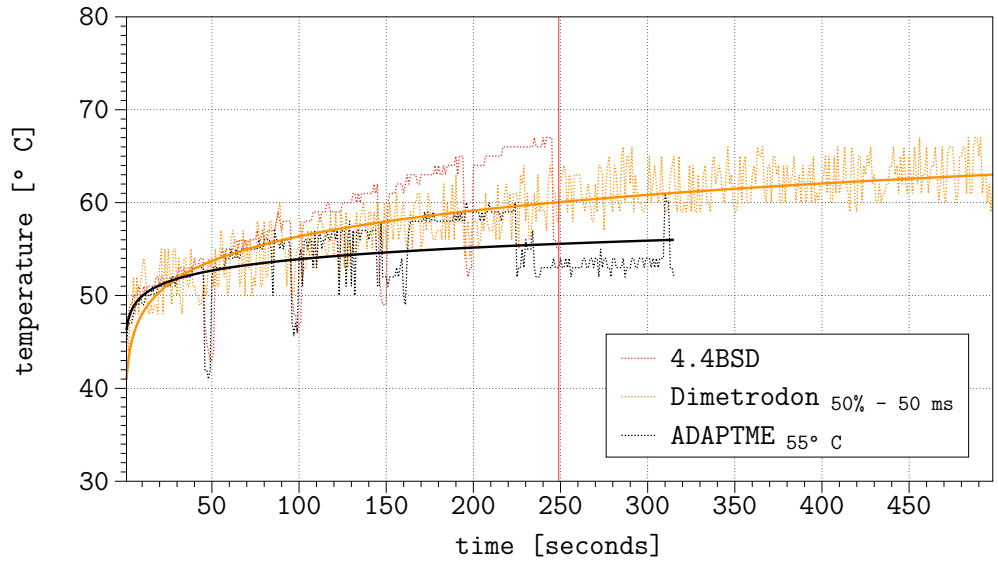

Figure 6.5: swaptions run results. Dimetrodon's average running temperature is comparable to that of FreeBSD/4.4BSD at the cost of a significantly higher execution time (∼ 2x). ADAPTME, on the other hand, keeps the average temperature under the set point of 55° C and imposes a small overhead on the execution of the workload. The vertical red line indicates the time when FreeBSD/4.4BSD finishes executing the benchmark application.

Dimetrodon was empirically configured so that the parameters would yield a an average temperature near that of ADAPTME's set point, without impacting too much on execution time. It was decided again to consider as a viable choice for its parameters a set of values by which the temperature was neither not too higher than ADAPTME's nor execution time too longer than ADAPTME's.

ADAPTME was configured so that the set point of the thermal-aware policy is 55° C. Being the only running application and being interested in focusing only on the thermal aspects of the work, it is set a goal for the application that would be met anyway (min heart rate: 1 heartbeat per second; max: 10^8 heartbeats per second) so that the performance-aware policy is actually ineffective.

In Figure 6.4 the obtained results are shown. As expected, FreeBSD/4.4BSD has the lowest execution time at the expense of temperature, which reaches 67°C. Observe how the subsequent runs in FreeBSD/4.4BSD causes the system to quickly heat and quickly cool down. During this spike the system activates cooling mechanisms such as fans to lower the temperature and avoid electronics break down, with consequent waste of power due to increase in fan speed. This is again com-

patible with the notion of "*run-to-idle*" scheduler. Dimetrodon, in this case, achieves its stated goal of lowering the temperature (by only 3°C), but since the system randomly injects idle cycles during execution, the system experiences an overall slowdown by a factor of 2. On the other hand, ADAPTME successfully lowers the average and peak temperature by 11°C, oscillating around its goal, while slowing down the application by a smaller factor of 1.28.

6.3.6 `swaptions`, multiple instances

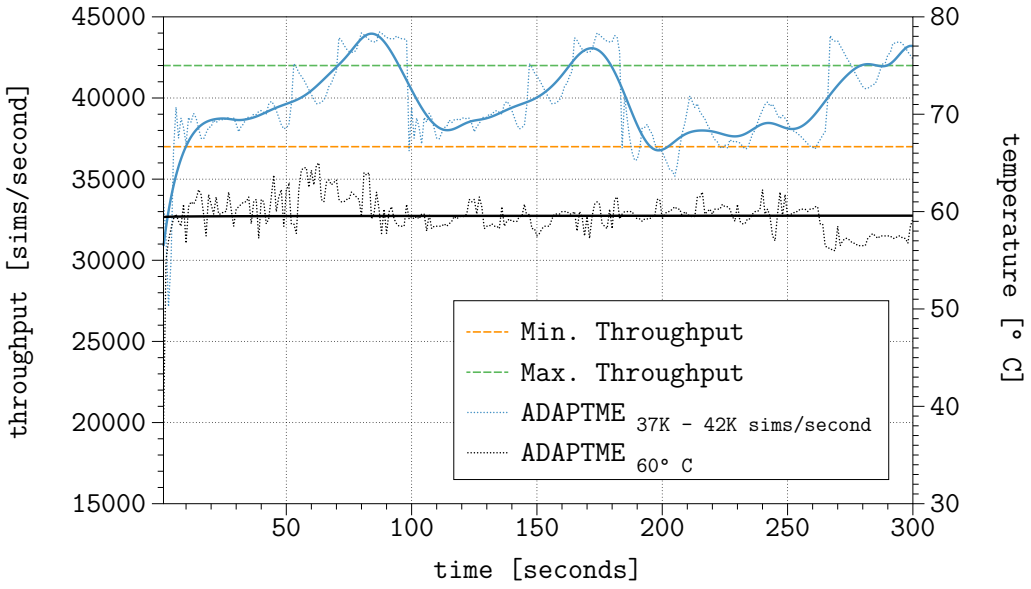

Figure 6.6: Multiple runs f swaptions. Heart rate of the second group is plotted as the blue line and is referred to the first y-axis (the left). It stays mostly inside the specified goal (min and max, respectively, are the dotted orange and green line). The temperature and the average temperature are plotted in black and referred to the second y-axis (the right)

In this experiment we test the interaction of the coupled thermal and performance aware policies.

In order to do so, we first run swaption application alone, and record the peak heart rate $\mathtt{hr}_{\mathtt{peak}}$ obtained by means of the HRM infrastructure. Then, we launch 4 instances of the same application; of course, the measured heart rate of each instance is $\mathtt{hr}_{\mathtt{peak}}/4$. We than specify two different goals: for the first group (in the HRM sense: see 2.4.1) of three applications, we set a goal that is *lower* than $\mathtt{hr}_{\mathtt{peak}}/4$, while we specify a goal which is *higher* than $\mathtt{hr}_{\mathtt{peak}}/4$ for

the group made of the other application. In this way, it is substantially favored the execution of one of the instances at the expenses of the others. Since s single instance of swaptions emits heartbeats at a rate of 1.2×10^5, it was decided to set as goal for the second group of 3.7×10^4 – 4.2×10^4 (min and max desired heart rate, respectively).

At the same time it was specified a temperature set point for the thermal-aware policy, in order to try to obtain the stated performance goal without violating thermal constraints and triggering the cooling infrastructure. As it was already saw (refer to Figure 6.5), the temperature reached by swaptions under FreeBSD/4.4BSD (*race-to-idle* approach where the highest observed peak temperature is obtained) is 67° C. So for the thermal-aware policy to be effective it must be stated a lower set point. In this experiment the set point is at 60° C.

Now, let's refer to Figure 6.6, where the heart rate of the second group is plotted with temperature of the CPU versus the execution time. The heart rate of the first group is not reported for the sake of readability; it suffices to say that it was in line with our expectations (i.e.: lower than $hr_{peak}/4$). It is seen that both the temperature and the performance goals are reached. Since it was seen in previous experiments that this is not the natural situation under which the system executes, the conclusion is that the two autonomic policies are effectively working towards the achievement of user's goals.

It is interesting to observe that the two sudden falls of hear rate in the graph are due to the combined actions of the two policies: what is happening inside the Operating System (OS) is that the performance policy is telling the thermal aware policy that since the upper bound has been surpassed, there is no reason to execute it again until the heart rate falls down again under that bound. This results in a sudden fall in heart rate, which is followed by a more gradual return to a normal condition.

6.4 Concluding remarks about experimental results

Table 6.2 summarizes the results of our experiments.

Table 6.2: Average measured temperature and standard deviation for applications from the PARSEC 2.1 Benchmark Suite expressed in Celsius degrees and runtime overheads expressed with respect to the 4.4BSD scheduler runtimes

Application	*Dimetrodon*			*ADAPTME*		
	Average [° C]	Std. Dev. [° C]	Overhead	Average [° C]	Std. Dev. [° C]	Overhead
blackscholes	51.16	3.45	2.02×	49.96	2.98	1.59×
ferret	58.02	5.01	2.35×	56.36	2.97	2.10×
fluidanimate	60.48	2.36	4.38×	58.86	3.20	3.31×
swaptions	59.00	4.60	2.00×	54.03	3.33	1.27×
x264	56.47	3.42	1.94×	49.25	3.70	1.61×

The thermal-aware policy was able to achieve at least the same temperature reduction measured with Dimetrodon, with the advantage of greatly reducing the runtime. From a wider point of view, temperature awareness allows some optimizations in the scheduling behavior and in the decision of where and when to introduce idle cycles, improving the overall effectiveness. Notice, for example in Figure 6.5, that ADAPTME allows to cap the temperature threshold faster; the policy recognizes that, at the beginning, the temperature is not critical and therefore it does not insert as many idle cycles as Dimetrodon does, effectively reducing overall execution time.

Table 6.2 reports into more detail the average temperatures with the standard deviations achieved with both Dimetrodon and the thermal-aware policy alongside with the runtime overhead with respect to the execution time observed under the 4.4BSD.

ADAPTME was evaluated with a workload combining four instances of the swaptions benchmark, each one composed of four threads. Three instances of swaptions were run freely while the fourth instance was run with a user-specified performance goal. The non-legacy instance of swaptions performance stays mostly within the user-specified performance window while the average temperature is very close to the system-specified target. . Hence, the thermal- aware policy acts on the remaining time sharing user threads to reach the system-specified temperature target.

This is the result of the combined efforts of the performance-aware policy, which adjusts the priorities of the four threads of the non-legacy instance, and of

the thermal-aware policy, that forces and prevents idling these threads whenever they are over-performing or under-performing

Chapter 7

Conclusions and future work

It was presented ADAPTME, a self-adaptive system able to tune both applications' and overall system's performance according to user-specified high-level goals. At the same time, it successfully and properly control processing cores temperatures, in compliance with a system-specified target. Our experimental results, based upon a state of art benchmark suite, and collected using a fully working extension of the FreeBSD 7.2 kernel running on contemporary hardware, show a proper control of both processing cores temperatures and non-legacy applications performance. Moreover, experimental results contain an extensive comparison between ADAPTME and Dimetrodon, a state of the art extension of the same operating system designed and implemented to preventively reduce average-case processing cores temperatures. The comparison highlights the advantages of ADAPTME, that results more flexible and outperforms Dimetrodon both in terms of average temperature and average throughput. This work also validates concepts and ideas expressed by the autonomic computing community regarding the efficacy of multiple Observe Decide Act (ODA) loops inserted at various levels in the OS, as well as the viability of multiple autonomic policies potentially concurring for opposite goals.

7.1 New monitors and adaptation policies: further developments

Further development on this work is already under its way. The first step is the addition of the adaptation manager envisioned in Section 3.2, so as to account for the presence of other autonomic policies in the system. One such policy is a power-aware policy, which is a policy capable of intervening on the power consumption of the system by means of intelligent scheduling and management of power states.

In order to extend the "knowledge of self" of the system, it is also sought the addition of improved thermal monitors, i.e. monitors capable of knowing the thermal condition of *portions* of the die, so as to work towards a thermal-aware policy that reduces the possible unbalances of thermal maps of the dies. Another monitor sought in the context of Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) is the performance counter monitor, which would be essential for power usage estimation and prediction.

Another major improvement would be the refinement of the model of the control theoretical system underlying the performance aware policy, since it was already known that it is a great simplification of the actual system.

7.2 Explicitly trading performance for temperature and vice-versa

In ADAPTME, the thermal-aware policy and the performance-aware policy are coupled together, and run asynchronously one with respect to the other. Since both policies are "best-effort" ones, it is sought a user-settable parameter that indicates how much the system should prefer the application of the former or of the latter, i.e. how much the system should care about temperature mitigation (possibly worsening tasks' performance) versus performance goal compliance (possibly worsening the thermal condition of the system). This way, an administrator can explicitly state what kind of "high-level" behavior the system is expected to

expose, which is a stated goal of the Autonomic Computing (AC) initiative.

Bibliography

[1] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2009.

[2] Christian Belady. The green grid data center power efficiency metrics: Pue and dcie. Technical report, teh green grid, October 2007.

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

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

[5] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.

[6] Parthasarathy Ranganathan. Recipe for efficiency: Principles of power-aware computing, April 2010.

[7] 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.

[8] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era, 2003.

[9] 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`.

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

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

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

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

[14] Kiril Schröder, Daniel Schlitt, Marko Hoyer, and Wolfgang Nebel. Power and cost aware distributed load management. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 123–126, New York, NY, USA, 2010. ACM.

[15] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The case for lifetime reliability-aware microprocessors. *Computer Architecture, International Symposium on*, 0:276, 2004.

[16] Robert F. Sullivan. Alternating cold and hot aisles provides more reliable cooling for server farms.

[17] Sriram Sankar, Mark Shaw, and Kushagra Vaid. Impact of temperature on hard disk drive reliability in large datacenters. In *DSN*, pages 530–537. IEEE, 2011.

[18] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface—scsi vs. ata. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 245–257, Berkeley, CA, USA, 2003. USENIX Association.

[19] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 8:1–8:15, Berkeley, CA, USA, 2008. USENIX Association.

[20] M. Santarini. Thermal integrity: A must for low-power ic digital design, Sept. 2005.

[21] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 2–13, New York, NY, USA, 2003. ACM.

[22] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: processor-level preventive thermal management via idle cycle injection. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 89–94, New York, NY, USA, 2011. ACM.

[23] US Environmental Protection Agency (EPA). Report to congress on server and data center energy efficiency: Public law 109- 431.

[24] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing, 2007.

[25] Stephen Shankland. Power could cost more than servers, google warns. Web, zdnet.com, December 2005.

[26] C. Belady. In the data center, power and cooling costs more than the it equipment it supports. Electronics Cooling, February 2007.

[27] Harvey M. Deitel. *An introduction to operating systems (2. ed.).* Addison-Wesley, 1990.

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

[29] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Hendrik F. Hamann, Alan J. Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In Diana Marculescu, Anand Raghunathan, Ali Keshavarzi, and Vijaykrishnan Narayanan, editors, *ISLPED*, pages 213–218. ACM, 2007.

[30] Jin Cui and Douglas L. Maskell. Dynamic thermal-aware scheduling on chip multiprocessor for soft real-time system. In Fabrizio Lombardi, Sanjukta Bhanja, Yehia Massoud, and R. Iris Bahar, editors, *ACM Great Lakes Symposium on VLSI*, pages 393–396. ACM, 2009.

[31] Wei-Lun Hung, Yuan Xie, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. Thermal-aware task allocation and scheduling for embedded systems. *CoRR*, abs/0710.4660, 2007.

[32] Eren Kursun, Chen yong Cher, Alper Buyuktosunoglu, and Pradip Bose. Investigating the effects of task scheduling on thermal behavior. In *In Third Workshop on Temperature-Aware Computer Systems (TACS'06*, 2006.

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

[34] M. Tim Jones. Inside the linux 2.6 completely fair scheduler. Technical report, 2009.

[35] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley, August 2004.

[36] The FreeBSD Project. Freebsd, 04 2012.

[37] Erven Rohou and Michael D. Smith. Dynamically managing processor temperature and power. In *IN 2ND WORKSHOP ON FEEDBACK-DIRECTED OPTIMIZATION*, 1999.

[38] Jürgen Becker Michael Hübner. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Springer-Verlag Gmbh, 1 edition, November 2010.

[39] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 171–, Washington, DC, USA, 2001. IEEE Computer Society.

[40] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81, New York, NY, USA, 1998. ACM.

[41] Thomas D. Burd, Student Member, Trevor A. Pering, Anthony J. Stratakos, and Robert W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 35:1571–1580, 2000.

[42] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In Shubu Mukherjee and Kathryn S. McKinley, editors, *ASPLOS*, pages 260–270. ACM, 2004.

[43] Jun Yang 0002, Xiuyi Zhou, Marek Chrobak, Youtao Zhang, and Lingling Jin. Dynamic thermal management through task scheduling. In *ISPASS*, pages 191–201. IEEE, 2008.

[44] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj K. Jha. Hybdtm: a coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 548–553, New York, NY, USA, 2006. ACM.

[45] Stefan Naumann, Markus Dick, Eva Kern, and Timo Johann. The greensoft model: A reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems*, 1(4):294 – 304, 2011.

[46] Shekhar Borkar. Design challenges for technology scaling, 1999.

[47] R Mahajan. Thermal management of cpus: A perspective on trends, needs and opportunities. Keynote at 8th Int'l Workshop on Thermal Investigations of ICs and Systems, 2002.

[48] Gunther et al. Managing the impact of increasing microprocessor power consumption., 2001.

[49] Heo S., Barr K., and K. Asanovic. Reducing power density through activity migration. In *In Proceedings of the International Symposium on Low-Power Electronics and Design.*, pages 217–222, New York, NY, USA, 2003. ACM.

[50] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, energy, and thermal considerations for smt and cmp architectures. In *HPCA*, pages 71–82. IEEE Computer Society, 2005.

[51] Pedro Chaparro, Grigorios Magklis, José González, and Antonio González. Distributing the frontend for temperature reduction. In *HPCA*, pages 61–70. IEEE Computer Society, 2005.

[52] Kevin Skadron, Tarek F. Abdelzaher, and Mircea R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *HPCA*, pages 17–28. IEEE Computer Society, 2002.

[53] Jayanth Srinivasan and Sarita V. Adve. Predictive dynamic thermal management for multimedia applications. In Utpal Banerjee, Kyle Gallivan, and Antonio González, editors, *ICS*, pages 109–120. ACM, 2003.

[54] Joachim Gerhard Clabes et al. Performance throttling for temperature reduction in a microprocessor. Patent, May 2006.

[55] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *ISCA*, pages 78–88, 2006.

[56] Massoud Pedram and Shahin Nazarian. Thermal modeling, analysis, and management in vlsi circuits: principles and methods. In *Proceedings of the IEEE*, 2006.

[57] Thidapat Chantem, X. Sharon Hu, and Robert P. Dick. Online work maximization under a peak temperature constraint. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, pages 105–110, New York, NY, USA, 2009. ACM.

[58] D Liu and C Svensson. Trading speed for low power by choice of supply and threshold voltages. *IEEE Journal of Solid State Circuits*, 28(1):10–17, 1993.

[59] 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.

[60] Vinay Hanumaiah, Sarma Vrudhula, and Karam S. Chatha. Performance optimal online dvfs and task migration techniques for thermally constrained multi-core processors. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 30(11):1677–1690, November 2011.

[61] Karin M. Abdalla and Robert J. Hasslen. Functional block level clock-gating within a graphics processor. U.S. Patent, Dec 2006.

[62] Pratyush Kumar and Lothar Thiele. Cool shapers: shaping real-time tasks for improved thermal guarantees. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *DAC*, pages 468–473. ACM, 2011.

[63] Min Bao, Alexandru Andrei, Petru Eles, and Zebo Peng. Temperature-aware idle time distribution for energy optimization with dynamic voltage scaling. In *DATE*, pages 21–26. IEEE, 2010.

[64] 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.

[65] Sushu Zhang and Karam S. Chatha. Approximation algorithm for the temperature-aware scheduling problem. In Georges G. E. Gielen, editor, *ICCAD*, pages 281–288. IEEE, 2007.

[66] Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen P. Boyd, and Giovanni De Micheli. Temperature-aware processor frequency assignment for mpsocs using convex optimization. In Soonhoi Ha, Kiyoung Choi, Nikil D. Dutt, and Jürgen Teich, editors, *CODES+ISSS*, pages 111–116. ACM, 2007.

[67] Frank Bellosa. Os-directed throttling of processor activity for dynamic power management. Technical report, 1999.

[68] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling cool: Temperature-aware workload placement in data centers. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5. USENIX Association, 2005.

[69] Raid Zuhair Ayoub, Krishnam Raju Indukuri, and Tajana Simunic Rosing. Temperature aware dynamic workload scheduling in multisocket cpu servers. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(9):1359–1372, 2011.

[70] Nikhil Gupta and Rabi N. Mahapatra. Temperature aware energy management for real-time scheduling. In *ISQED*, pages 91–96. IEEE, 2011.

[71] Thidapat Chantem, Xiaobo Sharon Hu, and Robert P. Dick. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. *IEEE Trans. VLSI Syst.*, 19(10):1884–1897, 2011.

[72] Shengquan Wang and Riccardo Bettati. Reactive speed control in temperature-constrained real-time systems. *Real-Time Systems*, 39(1-3):73–95, 2008.

[73] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.

[74] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[75] Toshihiro Hanawa, Toshiya Minai, Yasuki Tanabe, and Hideharu Amano. Implementation of isis-simplescalar. In Hamid R. Arabnia, editor, *PDPTA*, pages 117–123. CSREA Press, 2005.

[76] Pierre Michaud. Atmi 2.0 manual, 2009.

[77] Tom English, Ka Lok Man, Emanuel M. Popovici, and Michel P. Schellekens. Hotspot: Visualizing dynamic power consumption in rtl designs. In *EWDTS*, pages 45–48. IEEE, 2008.

[78] Marek Chrobak, Christoph Dürr, Mathilde Hurand, and Julien Robert. Algorithms for temperature-aware task scheduling in microprocessor systems. *Sustainable Computing: Informatics and Systems*, 1:241–247, 2011.

[79] Pedro Chaparro, Jose Gonzalez, Grigorios Magklis, Qiong Cai, and Antonio Gonzalez. Understanding the thermal implications of multicore architectures.

[80] Wei Chen, S. Toueg, and M.K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(1), 2002.

[81] Change research group website - http://www.changegrp.org/, Apr 2012.

[82] Filippo Sironi, Davide Bartolini, Simone Campanoni, Fabio Cancaré, Henry Hoffmann, Donatella Sciuto, and Marco Domenico Santambrogio. Metronome: Operating system-level performance management via self-adaptive computing. *DAC'12*, June 2012.

[83] W. S. Levine. *The Control Handbook*. CRC Press, 2nd edition, December 2010.

[84] 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.

[85] Venkatesan Packirisamy, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring speculative parallelism in spec2006. In *ISPASS*, pages 77–88. IEEE, 2009.

[86] http://www.cs.berkeley.edu/ pbailis/projects/dimetrodon/.

April 2, 2012

Document typeset with LaTeX