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: Prof. Filippo SIRONI

Tesi di Laurea di:

Riccardo Cattaneo

Matricola n. 755279

Anno Accademico 2011–2012

To the loving memory of Vincenzo Caglioti

Contents

1 Introduction **1**

1.1 Contemporary computing: trends and paradigms 3

1.1.1 Multicore computing 3

1.1.2 Cloud computing 7

1.1.3 Autonomic computing 9

1.1.4 Power efficient computing 12

1.2 Scheduling 15

1.2.1 Preliminary definitions 16

1.2.2 Problem statement 18

1.2.3 Batch, interactive and real-time scheduling 19

1.3 Problem Statement 21

1.4 Summary 22

2 State of the art **23**

2.1 Overview of major Major Free, Libre or Open Source Software (FLOSS) Operating System (OS)s and Kernels 23

2.1.1 Linux 23

2.1.2 Early Linux Schedulers 24

2.1.3 FreeBSD 27

2.2 Policies for power and energy efficiency 28

2.3 Dynamic Thermal Management techniques 29

2.3.1 Hardware based Dynamic Thermal Management (DTM) 31
2.3.2 Software based DTM 33
2.3.3 Thermal-aware scheduling (TAS) 35
2.4 Autonomic Operating System (AcOS) 41
2.4.1 Project goals 42
2.4.2 Heart Rate Monitor 43

3 Methodology 44
3.1 Motivation 46
3.2 Control Theoretical thermal and performance aware policy . 48
3.2.1 Derivation of priority update equation 51
3.2.2 Derivation of idle-time injection policy 53
3.3 Heart Rate Monitor 53
3.3.1 Definitions 54
3.3.2 Usage 56
3.4 Autonomic policies 56
3.4.1 Thermal-aware policy 56
3.4.2 Performance-aware policy 56
3.5 Explicitly trading performance for temperature and vice-versa 56

4 Implementation 57
4.1 4.4BSD scheduler 59
4.2 Heart Rate Monitor (HRM) 60
4.2.1 libhrm user space API 60
4.2.2 Implementation 60
4.2.3 Thermal aware policy implementation 63
4.2.4 Performance aware policy implementation 64
4.2.5 Heart Rate Monitor and ADAPTME 64
4.3 Benchmarking in a multicore environment: PARSEC 64

5 Results 65

6 Conclusions and future work **68**

A Code listings **70**

B Control theory introduction **71**

Bibliography **80**

List of Figures

1.1 Breakdown of total data center hardware and overheads costs of a representative Google datacenter [1]. 14
1.2 The model for Power Usage Effectiveness (PUE). 14

3.1 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). 47
3.2 The setting of the control problem 52

List of Tables

4.1 *libhrm* API[1,2] 60

List of Listings

List of Abbreviations

AC Autonomic Computing

AcOS Autonomic Operating System

AS Autonomic System

CFS Completely Fair Scheduler

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

CMP Chip Multi Processing

CPU Central Processing Unit

CRAC computer room air conditioner

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

ILP Instruction Level Parallelism

I/O Input/Output

I-Cache Instruction-Cache

IT Information Technology

MTTF Mean Time-To-Failure

NIST National Institute of Standards and Technology

OS Operating System

PID Process IDentifier

PUE Power Usage Effectiveness

QoS Quality of Service

RCSP Resource Constrained Scheduling Problem

SLA Service Level Agreement

SMP Simultaneous Multi Processor

SMT Simultaneous Multi Threading

TAS Thermal-aware scheduling

TID Thread IDentifier

TLP Thread Level Parallelism

Summary

Da dimetrodon This work focuses on reducing average-case processor operating temperatures, exploring the trade-offs between application perfor- mance and long-term thermal behavior through preventive thermal management. Our focus is on thread-level thermal management; once a thread is executing on a particular core, we want to control its thermal impact. Multicore-aware strategies, such as core migra- tion [11] as well as more complex thermal-aware thread schedule placement [9], are orthogonal to the problem we consider here but are potentially complementary to our goals. We focus solely on reducing temperature but also ensure that additional energy is not consumed by the CPU as a result. Dimetrodon1 is a software-level thermal management technique designed to assist in application-level proactive thermal manage- ment. We employ idle cycle injection, a scheduler-level mechanism to inject idle cycles of variable length into process execution, pro- viding responsive, fine-grained control, allowing individual threads to absorb substantial portions of the burden of cooling, carefully mitigating performance reductions. Per-thread policy control al- lows us to target only key heat-producing workloads as opposed to system-wide policies such as current dynamic voltage and fre- quency scaling (DVFS) mechanisms, which may unfairly penalize heterogeneous workloads [12].

Sommario

Chapter 1

Introduction

The way the semiconductor industry is keeping up in the recent years with the pace of change set by Moore's law, which states that the amount of transistors per chip roughly doubles every nearly two years as a consequence of the ever increasing request for improved performance of computing systems [2], has seen a sharp drift from the path that was followed until the early 2000s. As we reach the physical limitations 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 [3]. The power density of nowadays' microprocessors . 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 [4].

In addition, for some years now we are experiencing the explosion of the mobile computing era [3]. 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) [3]. 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 [5, 6]. 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 [5, 6].

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.

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

tems point of view, respectively).

1.1 Contemporary computing: 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: multicore architectures [4], cloud computing [7]and autonomic computing [5].

1.1.1 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)s transistors and consequently by increasing their clock speed [4]. 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 [4, 3]. 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) [3].

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 [3]. As already stated, architectural means to extract ILP have been implemented in the previous years [4]. 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 [4].

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) [3]. 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 [3]. 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 [3].

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 [4] (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 [4].

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* [8]. 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 Operating System (OS) 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 OSs: 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) [8]. Simultaneous Multi Threading (SMT) and fine grained multithreading technology, by which multiple threads' execution is interleaved in the *same* core [4], 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 hard-

ware, 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 [4] (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 [4] (in the case of directory-based caches). This implies scalability issues, in that it is not possible to add any given number of cores to a processor, since it may not be possible to implement an efficient communication means among all of them [4]. This problem becomes particularly important when one realizes that as applications are going to be designed from scratch to be multithreaded in order to fully take advantage of this new kind of architecture, the amount of cache coherence-related traffic on the processor bus is only expected to grow. Operating systems support may alleviate this problem (for example: by scheduling threads belonging to the same application to the same cores between different context switches) but may not cancel completely it.

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.

1.1.2 Cloud computing

The following is the definition of Cloud Computing as suggested by the U.S. National Institute of Standards and Technology (NIST) [9]:

"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

This form of computing service realizes the vision of *computing as a utility* [10, 11]. The novelty of this computing model is the possibility for anyone to consider computing power as like as any other form of public utility, like electricity, gas and water, available on demand, quickly enough to follow load surges and thereby eliminating the need to plan far ahead the provisioning of computing resources, thus paying just for the resources that are really needed for realizing a certain service [12, 13, 14, 15]. This removes the need for the typically relevant initial commitment in building the computing infrastructure able to serve the peak workload (which is the infrastructure typically commissioned when building a datacenter [1]) and allows a *pay-as-you-go* business model. Cloud computing is based on computing facilities called *clouds*, which are defined as "large pools of easily usable and accessible virtualized resources" (such as hardware, development platforms and/or services) [12, 10]. In particular, cloud computing services are realized by making a computing cloud accessible from the Internet and offering to the public a service of utility computing which promises reliable services delivered through next-generation data centers that are built

on compute and storage virtualization technologies [12, 11, 15] . A cloud computing provider can realize the illusion of unlimited computing and storage resources available on-demand, while cloud computing customers must pay only for the resources they are requesting, which can be adapted on a short-term basis and with no long-term commitment [12]. Another appealing feature of cloud computing is that consumers are assured that the cloud infrastructure is very robust and will always be available at any time [11], a characteristic that only comes with heavy investments in security and reliability of the whole infrastructure.

From a more technical point of view, considering as examples the cloud computing offers of Google, Amazon and Microsoft, very different in their peculiarities, but fundamentally the same under a more cursory view, we see that "clouds are clearly next-generation data centers with nodes virtualized through hypervisor technologies such as VMs, dynamically provisioned on demand as a personalized resource collection to meet a specific service-level agreement, which is established through a negotiation and accessible as a composable service via Web 2.0 technologies." [11].

The description of these datacenter underlines two important aspects of cloud computing:

- these are *state-of-art*, *shared* computing infrastructures. The efficiency of such data centers is way higher than any single datacenter that may be owned by the average cloud computing user, since they would typically need to over-provision it to keep-up with the rarely occurring peak-load events and the costs for installing a state-of-art infrastructure may probably be out of budget [15]. In other words, less computing resources need to be installed and maintained to allow for the same Quality of Service (QoS) for the same services, which is a main driving factor for the success of this paradigm. Data centers of these dimensions, shared by a large number of users, justify increased in-

vestments in energy efficiency, which is one major goals of the Green Computing initiative [15].

- such a big, distributed, complex and *dynamic* infrastructure benefits from improved automation of infrastructure and computing resources management. To materialize such idea, cloud computing services must be able to autonomously scale up and down, adapting the involved resources to the current workload and requested QoS and Service Level Agreement (SLA)s. In this framework, IBM research on Autonomic Computing [6, 16] is a valid means for increasing the efficiency and utility of such systems.

1.1.3 Autonomic computing

With the term Autonomic Computing, IBM describes in [6] 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 [6, 16] 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 [6, 16]. 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 [5] 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" [5, 6, 16, 6].

"Knowing" itself the system must have detailed knowledge of its components, status, capacity, connections and available resources, either in an exclusive or shared way.

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 way to exploit its resources in the most efficient possible way, moni-

toring 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 I/T resources to shrink the gap between the business or personal goals of our customers, and the I/T implementation necessary to achieve those goals without involving the user in that implementation.

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.

1.1.4 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 [3]. 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 [3]. The battery is often the largest and heaviest component of the system, so improved battery life implies smaller and lighter devices [3] 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 [3]. 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 [3].

Being efficient at consuming power has a three main advantages. The most obvious one is that reduced power consumption directly implies reduced running costs. Second, reduced power consumption leads to a 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 [3].

Thermal management, in particular, is becoming increasingly important due to the level of miniaturization of modern electronics and the in-

creased 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 [3, 7]. Studies, most notably concerning servers and hard-disk failures, have shown that operating 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 [17]. 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 [3, 18, 19]; similar statistics have also been shown over hard-disk lifetimes [17, 20, 21, 19]. Temperature directly affects also power consumption, clock latency and since processor leakage power increases exponentially with temperature, also CPU power consumption [22, 23]. At 90-nm-process nodes, leakage accounts for 25 to 40% of total power consumed [22]. At 65-nm-processes, leakage accounts for 50 to 70% of total power absorbed [22]. Moreover, a 15°C increase in temperature might causes delay of approximately 10 to 15% [22]. 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 [24].

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 [3, 7]. Up to 80% of data center construction cost is attributable to power and cooling infrastructure [1, 7], and chiller power, a historically dominant data center energy overhead, scales quadratically with the amount of heat extracted [7]. Research is ongoing in alternate cooling technologies (such as efficient liquid

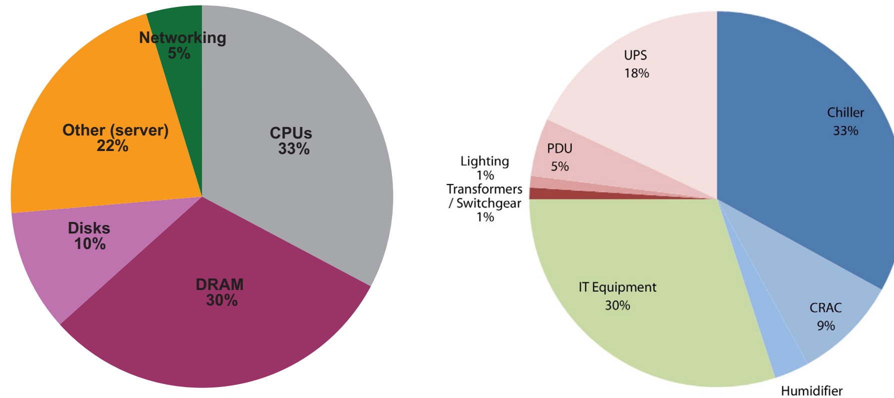

Figure 1.1: Breakdown of total data center hardware and overheads costs of a representative Google datacenter [1].

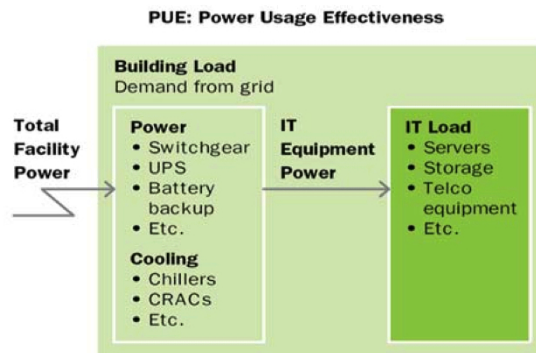

Figure 1.2: The model for PUE.

cooling), but it will still be important to be efficient about generating heat in the first place [3].

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) [25]. PUE is defined as the total facility power/IT equipment power, effectively measuring a form of overall data-center infrastructure efficiency; please refer to Figure 1.2.

Power management issues are only expected to be more and more pre-

dominant in the foreseeable future [3]. 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 development of the entire mobile/embedded industry [3]. 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 [26]. 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 [27] up to the point at which power and cooling cost might overtake hardware costs [28, 27, 29].

In Section 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).

1.2 Scheduling

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 [8]. 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* [8].

1.2.1 Preliminary definitions

In order to better understand the following section, we shall give here some preliminary definitions of interest for the scheduling problem [8].

The process is a basic concept for multiprogramming operating systems, as it defines the basic structure for managing code in execution. A process is fundamentally a container that holds all the information needed to run a program [8]. 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 [8].

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 operating system (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 [8].

The time that is assigned a process for running is called "quantum". On most systems this is a fixed amount of time (typically forced 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 requested 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 we report 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 are frequently 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 die 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, as creating and destroying threads is much faster than is for processes, applications that creates and destroys a large number of threads during their execution will gain considerable performance [8, 30]. 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.*

1.2.2 Problem statement

As we already exposed in 1.2, scheduling is the activity of choosing which process is going to be run in the next quantum of CPU time. We recall here one possible, and very general, definition of Resource Constrained Scheduling Problem (RCSP), given in [31, 32]. 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.

1.2.3 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 [8, 30].

Traditionally, a suggested classification of the possible workloads environments is the following [8, 30]

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 [31, 8, 30].

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 sched-

uler 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[33, 34, 35, 36]. This allows for a new goal to be introduced, namely

thermally constrained by which we indicate 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.

1.3 Problem Statement

As it was exposed throughout Chapter 1, thermal constraints in high performance computing environments are becoming a major issue for systems' designers which is tackled with different techniques that we will describe in much greater detail in Chapter 2.

In this work we propose an advancement in the context of operating system-based, local Dynamic Thermal Management (DTM) technique we called ADAPTME, a simultaneously performance and thermal aware scheduler aimed at server environments with general purpose (both batch and interactive) workloads. In order to validate our approach, we take as a reference the work of [24] and implement our system as a suitable patch for the same operating system the authors used.

1.4 Summary

We have introduced a number of concepts and motivations that are relevant to this thesis. A more thorough and comprehensive overview of 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 we shall introduce the basic techniques and the general ideas that underly the work. In Chpater 4, the system that has been designed and implemented will be dedescribed in great detail, and its experimental evaluation reported in 5.

Chapter 2

State of the art

2.1 Overview of major Major Free, Libre or Open Source Software (FLOSS) Operating System (OS)s and Kernels

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 we focus on the two most relevant works in this context, GNU/Linux and FreeBSD.

2.1.1 Linux

Linux is a Unix-like monolithic kernel developed by the open source movement since 1991[37]. It is the kernel most widely installed in open source operating systems [37]. 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.

2.1.2 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 [38]. The 1.2 Linux scheduler used a simple and fast circular queue for runnable task management that operated with a round-robin scheduling policy [38].

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) [38].

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 [38].

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 num-

ber 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 [38].

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 [38].

The CFS scheduler

The main idea behind the Completely Fair Scheduler (CFS) is to maintain balance (fairness) in providing processor time to tasks [38]. 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 [38].

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 [38]. 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 [38].

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 [38].

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 [38].

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 [38].

To date, CFS remains the default Linux scheduler.

2.1.3 FreeBSD

FreeBSD [39, 40] is another one of the major FLOSS kernel available.

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 [39]. 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)) [39].

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, the system switches to that thread immediately if the current thread is in user mode. Otherwise, the system switches to the higher-priority thread as soon as the current one exits the kernel citedesignfreebsd. The system tailors this *short-term scheduling algorithm* to favor interactive jobs by raising the scheduling priority of threads that are blocked waiting for I/O for one or more seconds and by lowering the priority of threads that accumulate significant

amounts of CPU time. Short-term thread scheduling is broken into two parts

Idle priority threads are only run when there are no time sharing or real-time threads to run. Real-time threads are allowed to run until they block or until a higher priority tasks is placed onto the multilevel feedback queues.

The ULE scheduler

2.2 Policies for power and energy efficiency

There is a wide spectrum of techniques that allow for power efficient computing to take place [24]. In this section we are going to focus 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 [18]), 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 [6].

Local techniques fall in two different and complementary categories: those that are implemented at the microarchitectural level [23] and those implemented at software level (either operating system level or programming language/paradigm level) [41]. They belong to the first category techniques such as DVFS, fetch throttling, and clock gating [42], which are standard features of modern microprocessors [24]. These techniques are par-

ticularly suitable for *reactively* reducing cores temperature, and reduce the burden of worst-case temperature management [43, 44, 45]. They don't try to *proactively* contrast the rise of the temperature. Software-level scheduling schemes [33, 46, 47], 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 Hyb-DTM [48], and programming paradigms/models such as GREENSOFT [49], complete the overview of currently available techniques. In the next section, we shall focus 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 [50, 51]. 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 cooling 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 [47]. 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 [33]. 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, Dynamic Thermal Management (DTM) techniques have been getting more popular for their low cost, flexibility [33] and stated goal of allowing designers to focus on average case rather than worst-case thermal conditions [43].

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

cro 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 we shall 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, we will focus on solutions that rely on OS support for thermal control.

2.3.1 Hardware based DTM

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 [33].

Microarchitecture-related DTM relevant researches are [43, 52, 53, 54, 55, 56, 23, 57, 58, 59].

Dynamic Voltage and Frequency Scaling (DVFS)

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 microprocessor so as to find a point in the configuration space that allows the system to reach a suitable thermal and power saving condition [47].

DVFS dynamically chooses the best tradeoff between power consumption and performance selecting a stable voltage supply/working frequency pair configuration [43, 60, 61, 59]. 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 [62, 52].

Research has traditionally focused on single core architectures [56, 23, 57], 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 [63].

Notably, in [64], 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 [43].

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 [65, 59, 43].

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 [4]. 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 [53, 43].

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 [60].

Instruction-Cache (I-Cache) throttling

I-Cache throttling allows the processor's fetch bandwidth to be reduced when the CPU reaches a temperature limit [43]. 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 based DTM

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 [33].

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 [66, 24] as a means to lowering CPU temperatures. In [67], 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 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. In [67], the authors address the problem of distributing idle time among different tasks at different voltage and frequency levels for energy minimization.

Core migration

Core migration [46], is a multicore-aware strategy by which "hot" and "cool" threads, as defined before, are run on different cores in a round robin fashion 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 section 1.1.1, one of the major challenges for operating systems schedulers aimed at multicore architectures is to find a way to achieve maximum parallelism while preserving data locality in caches as much as possible; obviously, running threads in a round-robin fashion 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 (TAS)

The main idea behind this kind of scheduling, which is operated at OS-level, is to execute jobs with different CPU usage profiles in order to induce variations in CPU temperature [43]. TAS realizes a kind of scheduling that has the explicit goal of keeping system's temperature below a given threshold [47, 35]. By means of an intelligent schedule, overall system temperature can be put under control.

This typically involves classifying running tasks as "hot" or "cold" , de-

pending on their relative degree of CPU-boundedness and Input/Output (I/O)-boundedness [68]. 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 [47, 64, 34, 53, 68, 35], TAS-specific heuristics [47, 68], optimal policies possibly obtained by means of approximate solutions [69, 34, 64, 70], task-related performance counters [71, 68], physical location of the machine [72] or CPU [73]. 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 [73, 72, 24], 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 [74, 75], 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 we shall better describe 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 usages. In recent years, due to increasing interest around the problem of characterization of workload, many different techniques and tools have been developed to model power profiles of applications. 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 [68] use a simple thermal model for characterizing the application, based on the work by [76] 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.

In [47], authors consider a simplified thermal model for a single core processor (stating that this is easily scalable to chip multiprocessors) based on [77] 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 [34] 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 [78] and SimpleScalar [79] 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 representative 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 [80] or HotSpot [81] as a thermal modeling framework.

Heuristic, formal and approximate solution approach to scheduling

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, the [40] operating system takes a scheduling decision every 10 ms. It is clear how much import it is to have an efficient scheduling algorithm, since it will be very frequently called during the execution of the system.

As authors in [82] claim, under their definition of TAS, the policy to find an optimal temperature-aware schedule has a 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 .

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 [82] and in [64], 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 scheduler which recurrently solves a complex dynamic programming problem that finds the best schedule for a set of hard real-time workloads given their deadlines, their recurring ar-

rival 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 Powre Based Thread Migration [83], 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 [64].

In another work, [47], 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 [47], performs better than MinTemp [36], 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.

Dimetrodon and the idle cycle injection technique

(rivedere articolo da cui questa frase è presa!) Traditional dynamic thermal management (DTM) techniques fo- cus on reducing worst-case thermal emergencies but do not con- tribute to lowering overall temperatures These techniques have many benefits such as increased reliability [25] and decreased chip pack- aging requirements [26] but are not designed to operate under nor- mal thermal conditions. In practice, these DTM mechanisms are only activated under extreme thermal conditions likely caused by catastrophic failures (e.g., cooling system problems). This work focuses on reducing average-case processor operating temperatures, exploring the trade-offs between application perfor- mance and long-term thermal behavior through preventive thermal management. Our focus is on thread-level thermal management; once a thread is executing on a particular core, we want to control its thermal impact. Multicore-aware strategies, such as core migra- tion [11] as well as more complex thermal-aware thread schedule placement [9], are orthogonal to the problem we consider here but are potentially complementary to our goals. We focus solely on reducing temperature but also ensure that additional energy is not consumed by the CPU as a result. Dimetrodon1 is a software-level thermal management technique designed to assist in application-level proactive thermal manage- ment. We employ idle cycle injection, a scheduler-level mechanism to inject idle cycles of variable length into process execution, pro- viding responsive, fine-grained control, allowing individual threads to absorb substantial portions of the burden of cooling, carefully mitigating performance reductions. Per-thread policy control al- lows us to target only key heat-producing workloads as opposed to system-wide policies such as current dynamic voltage and fre- quency scaling (DVFS) mechanisms, which may unfairly penalize heterogeneous workloads [12].

[quelli di dimetrodon fanno questo cliam rispetto a dvfs, noi non l'abbiamo

verificato del 30 per cento di riduzione rispetto a dvfs su temp]

2.4 Autonomic Operating System (AcOS)

riferirsi a change quali sono gli obbiettivi di caos autonomic etc etc finora prestazioni adesso anche temperatura **SANTA** questa sezione è completamente copiata da dbb X(la riscrivo stando sulla stessa linea? la levo? la restringo? This thesis has been developed within the Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE) research group, at the Dipartimento di Elettronica e Informazione (DEI) of the Politecnico di Milano. CHANGE group is dedicated to the creation of a self-aware computing system by mainly operating on any kind of modern computing device: from mobile devices and desktops to servers, mainframes and huge computing facilities. The aim of the project is to demonstrate that any of these computing systems can be enhanced by augmenting them with an autonomic layer able to continuously ensure optimal performance and simplified management of the system. Within this context, the notion of performance is extended beyond the mere idea that the faster, the better, but it comes to include objectives such as the 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.

To be able to realize this vision, the CHANGE group proposes a model for autonomic computing systems that is based on the ODA control loop (see Subsection 1.2.2), which should be present at multiple levels within the system. At a lower level, the single system components can benefit from autonomic management via internal ODA loops and, at a higher level, a broader control loop, aware of the system status as a whole, should be in charge of orchestrating the different subsystems towards the maximum performance (in the broad sense explained above). The rest of this Section

contains a presentation of the goals of the CHANGE group and a proposal of an approach for realizing them.

2.4.1 Project goals

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

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

to improve the autonomic features once the autonomic base system in the OS layer will be ready.

Thus, the OS is the glue between the hardware and the applications and it is possible to work with relative ease at this level; therefore, the operating system layer is the one chosen for embedding the main control loop in charge of marshaling all the (autonomic and non autonomic) system components in an adaptive and intelligent way. For this reason, the current goal of the CHANGE group is to work at the OS level in order to build a solid platform on which the development of a complete autonomic framework can be made possible. To realize this goal, the CHANGE group is creating a research operating system called Autonomic OS (AcOS), which gains its first partial implementation with the work proposed in this thesis.

2.4.2 Heart Rate Monitor

copia spudoratamente ad dac e da icac **SANTA** cito Metronome e spiego un pò cosa e, perchè serve, il legame con l'Autonomic Computing (AC), come funziona (libreria userspace+codice kernel space) e a grandi linee come è implementato. Quindi spiego perchè serve per il lavoro di FreeBSD

Chapter 3

Methodology

[occhio a confronto con soft-real time] 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 we decided to further the development of 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 we mean the property by which the scheduler guarantees 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 2.4.2) or those that have not expressed any goal. In case it is not possible to satisfy the goals of all the Heart Rate Monitor (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.

We see this feature 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 (an in particular at scheduling level), we decided to focus on *idle cycle injection*. Different reasons led us to take 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 recompute priorities based upon a relatively compact 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 [24]) implemented with success this technique on commodity operating systems, making them a good benchmark for comparing our solution to the state-of-art.

In the following sections we shall describe in grater detail the theoretical aspects of this work. In section 3.2, a brief recall of Control Theory and its application to this work is given to understand the claim of the stability of the system. HRM is then presented in section 3.3 as a means for the autonomic system to "know itself" . The two concurring policies composing our TAS algorithm is presented in 3.4. A brief recall on the practical aspects of this work is then given in 3.5.

3.1 Motivation

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. While this behavior has been traditionally considered optimal, nowadays computing environments may 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 heating and thus, on average, to lower the average running temperature. As already pointed out in section 1.1.4, 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 3.1 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 don't, 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; we believe that a similar solution can be applied to desktop and server systems, allowing users to provide applications performance goals and applications to signal execution progresses or latencies.

We implement two different polices, one thermal-aware and one performance-aware, the combination of which results in ADAPTME, a novel thermal

Figure 3.1: 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).

and performance aware scheduling infrastructure based on a popular general purpose scheduler, 4.4BSD, implemented in the FreeBSD Operating Sysem. Although we tried not to introduce significant changes in the scheduling infrastructure, we observed that the performance-aware policy would have possible caused system instability (for example: task starvation): for this reason, we implemented the policy based on notions of Control Theory so as to tackle this problem from a stability point of view. We outline the reasoning in the next section while a more thorough understanding of the stability problem is given in appendix B.

3.2 Control Theoretical thermal and performance aware policy

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

These two policies activates 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 the effects of the performance-aware policy do so only during priority recomputation (which instead occurs, by default, only after 10 context switches). Moreover, the effects of the thermal policy are *enforced*, meaning that the temperature mitigation goal will be reached, no matter what, while the performance policy is a "*best effort*" one, meaning that the scheduler will do its best to achieve the performance goal but it will not guarantee to do so. The thermal-aware policy monitors the temperature of each processing core and exploits idle cycles injections to slow down tasks whenever a target temperature is surpassed or near to do so, thus allowing the processing cores to activate low-power modes and producing less heat which in turn allows for the cooling system to stay turned off, which is one of the goals of this work.

This work is inspired by the thermal-unaware scheduler by [24]. Conversely to that work, we 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. 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 3.1.

$$T_i(k+1) = T_i(k) + \mu_i \cdot \mathtt{idle}_i(k) \tag{3.1}$$

$\mathtt{idle}_i(k)$ 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%, and μ_i is an unknown parameter. The control-theoretical system, designed as an adaptive deadbeat controller [84], computes $\mathtt{idle}_i(k)$ 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 [84] [ma quest e' hero a prescindere dal system 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.

[qualcosa sulla stima di μ_i? quali condizioni deve soddisfare affinche' si possa dimostrare la stabilita' del sistema? qualcosa sul modello in se, per esempio sulla sua relativa semplicita'?] [dire esplicitamente che r e' da hrm]

The performance-aware policy makes use of applications performance goals (i.e., user-specified throughput metrics) to adapt threads priorities

increasing or decreasing the amount of processing cores time threads are assigned to. These performance goals are expressed by means of a flexible performance goals infrastructure implemented at Operating System (OS)-level called HRM. We will more thoroughly describe the main features of this component in section 3.3.

The performance-aware policy features another control-theoretical mechanism to drive applications priority; the policy requires user-specified applications to signal performance goals and applications progresses. We 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 $r_i(k)$, 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. Since this scheduler fits into the soft real-time category of scheduling infrastructures, this is not regarded as an issue.

The model of the system is assumed to be the one reported in Equation 3.2.

$$r_i(k+1) = r_i(k) + \eta_{i,j} \Delta priority_{i,j}(k)(2) \quad (3.2)$$

$\Delta priority_{i,j}(k)$ 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 4.1 .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 should be transferred to the output rate. The same analysis holds in this case and, if η_i is known, the set point signal will be attained and the temperature will be kept below the reference level [84]. 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.

3.2.1 Derivation of priority update equation

Here we derive 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 we derive 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 3.2.

We 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 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

$$S : r(k+1) = r(k) + \mu \Delta p(k) \tag{3.3}$$

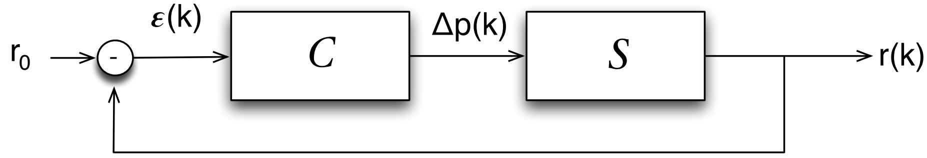

Figure 3.2: The setting of the control problem

Now we explicit the delay operator and factor out

$$z \cdot r(k) = r(k) + \mu \Delta p(k) \quad (3.4)$$

$$(z-1) \cdot r(z) = \mu \Delta p(k) \quad (3.5)$$

$$S : \frac{r(z)}{\Delta p(k)} = \frac{\mu}{z-1} \quad (3.6)$$

Then, we extract the basic equation for the loop transfer function and constrain it to be the unitary delay operator:

$$\text{loop} : \frac{C \cdot S}{1 + C \cdot S} = \frac{1}{z} \quad (3.7)$$

$$\frac{C \cdot \frac{\mu}{z-1}}{1 + C \cdot \frac{\mu}{z-1}} = \frac{1}{z} \quad (3.8)$$

$$C \cdot \mu = \frac{z - 1 + C \cdot \mu}{z} \quad (3.9)$$

$$C \cdot \mu \cdot z - C \cdot \mu = z - 1 \quad (3.10)$$

$$C \cdot \mu \cdot \cancel{(z-1)} = \cancel{z-1} \quad (3.11)$$

$$C \cdot \mu = 1 \quad (3.12)$$

$$C = \frac{1}{\mu} \quad (3.13)$$

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

$$\Delta p : \Delta p(k) = C \cdot \epsilon(k) \quad (3.14)$$

$$\epsilon(k) = r_0 - r(k) \quad (3.15)$$

$$\Delta p(k) = \frac{1}{\mu} \cdot (r_0 - r(k)) \quad (3.16)$$

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

$$\Delta r(k+1) = r(k) + \mu \cdot \delta p(k) \tag{3.17}$$

$$\Delta r(k+1) = r(k) + \cancel{\mu} \cdot \frac{1}{\cancel{\mu}} \cdot (r_0 - r(k)) \tag{3.18}$$

$$\Delta r(k+1) = \cancel{r(k)} + r_0 - \cancel{r(k)} \tag{3.19}$$

$$\Delta r(k+1) = r_0 \tag{3.20}$$

Which is exactly the definition of deadbeat controller.

3.2.2 Derivation of idle-time injection policy

3.3 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". 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 [85]. 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.

3.3.1 Definitions

In this section we provide 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 [cita dbb], as a measure of latency and a measure of contention [cita teo]. When heartbeats are employed for throughput means, we define a *hotspot* as 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*.

3.3.2 Usage

3.4 Autonomic policies

3.4.1 Thermal-aware policy

lettura temperatura, scheduling dell'idle e time wrt performance

3.4.2 Performance-aware policy

hrm per tirare fuori il QoS dell'applicazione

3.5 Explicitly trading performance for temperature and vice-versa

The thermal-aware policy and the performance-aware policy are then coupled together, with the latter providing more information to the former; the performance-aware policy marks those applications whose throughput can be further decreased and those applications whose throughput must be increased. Thanks to this distinction the thermal-aware policy injects idle cycles only when processes are not in critical conditions. [pararemtro tra 0 e 1 per variare sistema di controllo e prestazione]

Chapter 4

Implementation

...

We implemented ADAPTME in the FreeBSD 7.2 kernel, modifying the operations of the 4.4BSD scheduler1 to make a fair comparison with Dimetrodon, which was implemented on the same release of the operating systems. The 4.4BSD scheduler is augmented with the thermal-aware policy and the performance-aware policy. The thermal-aware policy consists of a set of high-priority kernel threads to regularly monitor the temperature of each core and a high-priority kernel thread implementing the control-theoretical system described in Section 3.1. Just like the thermal-aware policy, the performance-aware policy is made up of a set of high- priority kernel threads to compute applications performance and implement the control-theoretical system described in Section 3.1. 4.1 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. 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. Multiple threads on the same run queue are managed in a round robin fashion and are assigned a static quantum of time [12]. The 4.4BSD scheduler is based on a multilevel feedback queues infras-

tructure; threads migrate among run queues according to their changing scheduling priority. Higher priority threads preempt lower priority threads whenever they are added to a run queue. Since we were extending an existing scheduling infrastructure we put a lot of effort into preserving all the desirable properties coming with the 4.4BSD scheduler (e.g., non-starvation, priority decay). 1FreeBSD 5.1 kernel was the first release supporting two dif- ferent schedulers, namely 4.4BSD [12] and ULE [15]. Even tough the ULE scheduler is the default choice since FreeBSD 7.1 kernel, we modified the 4.4BSD scheduler to compare ADAPTME with Dimetrodon.

The performance-aware policy is an extension of the sched- uling infrastructure acting in a decoupled fashion. Threads priorities, which are updated at a constant rate, are adjusted using an additive term priorityi,j for each thread j of application i, where the application is a non-legacy one. This operation migrates threads from the current run queue to an- other one, advantaging or disadvantaging threads according to their measured performance and performance goals. In the 4.4BSD scheduler, the priority of a thread indi- cates 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 force idle if it is over performing or prevent idle 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. 4.2 Thermal-Aware Policy Conversely to the

performance-aware policy, the thermal- aware policy acts in coordination with the 4.4BSD sched- uler. When the scheduler chooses the next thread to run, it decides whether to run it or to run the idle thread pre- empting 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 prevent idle. The idle thread is also scheduled if the thread that is going to be preempted is marked with force idle. A thread is considered system critical if it is a bottom-half kernel thread (e.g., interrupts) or a top-half kernel threads; moreover, system critical kernel threads cannot be marked with neither force idle nor prevent idle. Given this operation mode, the thermal-aware scheduler is subsumed by the performance-aware scheduler; hence, per- formance goals take the lead with respect to thermal con- straints. To drive the natural trade-off between the thermal- aware policy and the performance-aware policy we further extended the process of choosing the next thread to run with a probabilistic method. The probabilistic choice is involved whenever the outputs of the control-theoretical systems con- flict.

4.1 4.4BSD scheduler

multilevel feedback queue termine di paragone porting di hrm (strutter base, attach e detach con syscall, no procfs, scambio dati con memory mapping

[starvation problem:decay cresce in maniera monotona noi incidano sul termine per cui diamo piu tempo]

le modifiche dove sono state fatte? perche' non avete lavorato sul quanto dinamico? problema di variare la semantica del quanto di tempo, riportare l'equazione di aggiornamento della priorita e grafo di chiamate di update_priority come da documentazione

Table 4.1: *libhrm* API[1,2]

Function	Description
`hrm_attach(int gid, bool_t consumer)`	Attach the current task to the group identified by `gid` as either a producer or consumer
`hrm_detach()`	Detach the current task
`hrm_set_min_heart_rate(uint32_t min_heart_rate)`[3]	Set the minimum heart rate in the user–defined performance goal
`hrm_set_max_heart_rate(uint32_t max_heart_rate)`[3]	Set the maximum heart rate in the user–defined performance goal
`hrm_set_window_size(size_t window_size)`[3]	Set the window size in the user–defined performance goal
`hrm_get_min_heart_rate(uint32_t *min_heart_rate)`	Get the minimum heart rate from the user–defined performance goal
`hrm_get_max_heart_rate(uint32_t *max_heart_rate)`	Get the maximum heart rate from the user–defined performance goal
`hrm_get_window_size(size_t *window_size)`	Get the window size from the user–defined performance goal
`hrm_get_global_heart_rate(uint32_t *global_heart_rate)`	Get the global heart rate from the performance measure
`hrm_get_window_heart_rate(uint32_t *window_heart_rate)`	Get the window heart rate from the performance measure
`int heartbeat(uint64_t n)`[3]	Emit `n` heartbeats

[1]Every function receive an additional parameter of type `hrm_t *` pointing to the underlying 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

4.2 Heart Rate Monitor (HRM)

4.2.1 libhrm user space API

libhrm, instrumentaione delle applicazioni di parsec, problema della definizione del significato dell'HR che è una metafora

lettura temperatura - cita manuali intel pezzi di codice rilevanti

4.2.2 Implementation

The implementation of HRM consists of two partitions, the user-space partition and the kernel-space partition. The former comprises a library, namely `libhrm` apps, exposing the API for both applications and systems developers; the API's basics are reported in Table 4.1. While the APIs functions for applications developers grant the ability to instrument applications, providing a way to specify performance goals and signal progresses, the APIs functions for systems developers are meant to retrieve applications performance measures and performance goals.

The API exposes two functions, hrm attach and hrm detach, to attach the current task to the group identified by a GID as ei- ther a producer/-consumer (i.e., application) or a consumer (i.e., adaptation policy) and to

detach the current task. Two functions, hrm set active and hrm set inactive, are implemented to ei- ther set active or inactive the current task: a task is said to be ac- tive if it is executing the hotspot, inactive otherwise. These two states prove to be useful to maintain performance measures3 in programs using spawn kill parallelization (e.g., x264 in the PARSEC 2.1 benchmark suite), in which there is no guarantee that at least one active task is always alive. Different applications may be concerned with either long- or short-term trends. There- fore, the API exposes both hrm get global rate, to catch long- term trends through the average heart rate over the whole execu- tion time, and hrm get window heart rate, to catch 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 ac- count for; the timer period controls how often performance mea- sures are updated. The window size and the timer period can be set through hrm set window size and hrm set timer period respectively. Two additional functions, hrm set min heart rate and hrm set max heart rate, are exposed to adjust performance goals, which are defined as a desired heart rate range. Other func- tions are available to retrieve performance goals and performance goals related parameters. The most important APIs functions are heartbeat and heart- beatN. Calls to these functions are inserted within the hotspot of a program to signal progresses by incrementing the summation of heartbeats either by 1 or by a generic integer value. The kernel-space partition of HRM consists of an API that mimics a subset of the functions described above, and the core of the active monitor. The implementation of HRM extends Linux in few key places: it introduces include/linux/hrm.h and kernel/hrm.c, which constitute the core of the active monitor, and modifies in- clude/linux/sched.h and fs/proc/base.c, which respectively expose HRM to the kernel-space and the user-space. Figure 2(a) shows the globally accessible list of groups

at the very base of the imple- mentation of HRM. The list of groups can be read in parallel and written serially by hrm attach and hrm detach function calls; to guarantee correctness, the list of groups is protected by a read-/write lock. Each group is provided with a set of memory pages devoted to heartbeats count and a memory page dedicated to performance measures and performance goal. The amount of memory pages to store heartbeats is a compile time tunable parameter. Memory pages are shared between the kernel-space and the user-space to reduce the overhead in accessing the information as much as possible. More specifically, the content of memory pages devoted to heartbeats count is the most critical to HRM since it can be concurrently accessed at a high rate by both kernel-space tasks and user-space tasks. A way to avoid overheads and concurrency issues consists in splitting the heartbeats count in a set of per-task heartbeats counts; hence, function calls to both heartbeat and heartbeatN reduce to an atomic variable increment. The amount of heartbeats counts stockpiled in memory pages is architecture dependent since they are cache line aligned. The implementation of HRM instantiates standard-sized memory pages of 4 Kbytes and x86 and x86-64 microprocessors feature cache lines of 64 bytes: this implies that each memory page can contain up to 64 heartbeats counts. Figure 2(b) shows the organization of the memory pages devoted to heartbeats count focusing on tasks accessing dedicated cache line aligned heartbeats counts. Different applications and adaptation policies may be concerned with either long- or short-term trends. Therefore, the 64 bytes of the memory page dedicated to performance measures and performance goal contain both a global heart rate, which accounts for the whole execution of a group and catches long-term trends, and a window heart rate, which accounts for the execution of a group over a time window and catches short-term trends. The global heart rate and the window heart rate are respectively computed according to Equation 1 and Equation 2. In the Equations,

g indicates the group, t indicates the current time, t0 indicates the time at which the group was created, and tw indicates the time at which the window started. The performance measures are asynchronously updated in kernel- space in the context of a High-Resolution (HR) timer.

?The second chunk of 64 bytes of the memory page dedicated to performance measures and performance goals contain a minimum heart rate and a maximum heart rate to define a performance goal through a heart rate range. Other available parameters are the win- dow size and the timer period; the latter sets the frequency at which performance measures are updated, while the former sets the win- dow size expressed in timer periods. Figure 2(c) shows the organi- zation of the memory page dedicated to performance measures and performance goal; each task accessing these information maps the whole memory page. The implementation of HRM integrates with the Linux kernel configuration system, allowing compile time customizations. The number of memory pages HRM uses for heartbeats counts is a compile time tunable parameter (default, 1 memory page, up to 16 memory pages). Such a limit is completely arbitrary, it is indeed possible to further increment this number by modifying a single value. The timer period at which the performance measures are updated is a compile time tunable parameter too and it ranges from 1 ms to 1 s (default 100 ms); moreover, this is also per group runtime tunable parameter.

4.2.3 Thermal aware policy implementation

tu non stia andando a bloccare un kthread e non ci siano settate le flag bloccami e non bloccami parametro che dice preferisci le performance preferisci la temperatura

4.2.4 Performance aware policy implementation

la parte di performance oltre a fissare il delta prio ok noi andiamo ad are consigli sulla parte e termal e′ meglio che non blocchi e′ meglio che lo blocchi

4.2.5 Heart Rate Monitor and ADAPTME

4.3 Benchmarking in a multicore environment: PARSEC

Chapter 5

Results

. . . We evaluated ADAPTME in a commodity operating sys- tem, namely FreeBSD [19] 7.2, with real-world parallel work- loads from the PARSEC 2.1 Benchmark Suite [3]2. 5.1 Thermal-Aware Policy Evaluation We tested the thermal-aware policy of ADAPTME 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. The same setup was used to per- form tests with Dimetrodon, which was patched to work on x86 64. ADAPTME was configured to run high-priority ker- nel threads to measure temperatures every 100 ms while the threads implementing the control-theoretical systems were run every 500 ms. We evaluated the thermal-aware policy of ADAPTME un- der a subset of the applications of the PARSEC 2.1 Bench- mark Suite, each run consisting of ten consecutive execu- tions of the same application registering the machine ther- mal profile. The same experiments were repeated with the 4.4BSD

scheduler, Dimetrodon, and the thermal-aware pol- icy of ADAPTME. Dimetrodon parameters (i.e., the prob- ability to inject idle cycles and idle time) were tuned for each different application to obtain a reasonable temper- ature reduction with respect to the one obtained with the 4.4BSD scheduler. The thermal-aware policy of ADAPTME was configured with a target temperature close to the aver- age temperature measured when Dimetrodon was executed. Figure 2 shows that the proposed 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, tempera- ture 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 2b, 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. Table 1 reports the average temperatures with the standard devia- tions 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 scheduler. During our extensive experimentation, which was neces- sary to tune Dimetrodon parameters and to execute PAR- SEC 2.1 Benchmark Suite applications with reasonable per- formance, we were not able to verify the runtime model pro- posed by Bailis et al. [1]. We argue that this is due to the multithreaded applications we employed, that are not di- rectly comparable to the multiprogrammed workloads made up of singlethreaded applications from the SPEC CPU2006 Benchmark Suite [18] used to verify the model. 5.2 Adaptive Performance and Thermal Man- agement Evaluation We tested the full system combining the thermal-aware policy and the performance-aware policy of ADAPTME on a different entry-level midtower server equipped with a single Intel Core i7-870 quad-core proces-

sor running at 2.93 GHz with a nominal maximum Thermal Design Power (TDP) of 130 W, 4 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. We evaluated ADAPTME 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 per- formance goal, expressed as the number of Monte Carlo sim- ulations per second. As shown in Figure 3, the non-legacy instance of swaptions performance stays mostly within the user-specified performance window while the average tem- perature is very close to the system-specified 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. Hence, the thermal- aware policy acts on the remaining time sharing user threads to reach the system-specified temperature target.

tabellina delta picco picco durante l'esecuzione

Chapter 6

Conclusions and future work

. . . We have presented ADAPTME, a self-adaptive system able to tune performance according to user-specified perfor- mance goals and to properly control processing cores tem- peratures in compliance with a system-specified target. Our experimental results, collected using a fully working exten- sion of the FreeBSD 7.2 kernel running on contemporary hardware, show a proper control of both processing cores temperatures and non-legacy applications performance. More- over, experimental results contain an extensive comparison between ADAPTME and Dimetrodon, a state-of-the-art ex- tension 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 Di- metrodon both in terms of average temperature and average throughput.

is similar to... is complementary to...

Aggiungere nova politica a thermal e performance che e' esplicitiamente energy/power aware e che leveragi -> To know which job will be hot or cool for the hotspot, we develop a highly efficient online temperature estimator, leveraging the performance counter-based power estimation [Isci and Martonosi 2003; Joseph and Martonosi 2001; Kumar et al. 2006], com-

pact thermal modeling [Skadron et al. 2003], and a fast tempera- ture solver [Han et al. 2006]. We implemented the estimator for a Pentium 4 processor, although our general methodology is applicable to other processors, such as CMPs.

"complicare" il modello del controllore

[il confronto fra noi e dimetrodon e inarticolate il particolare che il trade-off 30 percento temperatura poi dvfs si sposta e si caratterizza in bvase al workload legacy o non legacy]

Appendix A

Code listings

…

Appendix B

Control theory introduction

…

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

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

[4] 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.

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

[6] 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`.

[7] 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.

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

[9] Timothy Grance Peter Mell. The nist definition of cloud computing. Web article, September 2011.

[10] M. Armbrust et al. Above the clouds: A berkeley view of cloud computing. Technical Report 28, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, February 2009.

[11] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. *CoRR*, abs/0808.3558, 2008.

[12] Davide Basilio Bartolini, Matteo Carminati, Riccardo Cattaneo, Giuseppe Chindemi, and Santo Lombardo. Cloud computing and self-adaptation. *Proceedings of Advanced Topics of Sooftware Engineering Class, Politecnico di Milano*, 2011.

[13] Robert L. Grossman and Yunhong Gu. On the varieties of clouds for data intensive computing., 2009.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zahari. A view of cloud computing, 2010.

[15] Rajkumar Buyya, Anton Beloglazov, and Jemal H. Abawajy. Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges. *CoRR*, abs/1006.0308, 2010.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

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

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

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

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

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

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

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

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

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

[53] 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.

[54] 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.

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

[56] 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.

[57] 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.

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

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

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

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

[72] 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.

[73] 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.

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

[75] 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.

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

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

[78] 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.

[79] 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.

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

[81] 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.

[82] 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.

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

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

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

March 28, 2012

Document typeset with LaTeX