Winter 12-15-2013

Dynamic Thermal and Power Management: From Computers to Buildings

Yong Fu
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Chenyang Lu, Chair
Ron K. Cytron
Christopher D. Gill
Humberto Gonzalez
Anne Holler
Raj Jain

Dynamic Thermal and Power Control for Computing Systems

by

Yong Fu

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

December 2013
Saint Louis, Missouri

Contents

List of Figures

List of Tables

Acknowledgments

I would like to sincerely thank my advisers, Professor Chenyang Lu, whose guidance helped me through my Ph.D career. Chenyang has always been a source of not only inspiration but also support. The ideas presented in this paper, totally or partially, arise from discussions with Chenyang.

I would also want to thank Professor Xenofon D. Koutsoukos and Dr. Nicholas Kottenstette, who not only provided valuable knowledge of control theory but also partially shaped my research.

My internship at VMware offered me a wonderful opportunity to experience work and research under industry environment. I greatly thank my mentor, Dr. Anne Holler for making my internship a very enjoyable and rewarding experience.

Finally, I am grateful to have opportunity to collaborate with many colleagues who support my research : Mo Sha, Chengjie Wu, Abu Sayeed Saifullah, Sisu Xi, Dr. Yixin Chen, Dr. Humberto Gonzalez, Dr. Anna Leavey, Dr. Weining Wang, Bill Drake, Andrew Kutta, Dr. Pratim Biswas and Craig England.

Yong Fu

Washington University in St. Louis
December 2013

To Yan and Claire

ABSTRACT OF THE DISSERTATION

Dynamic Thermal and Power Control for Computing Systems

by

Yong Fu

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2013

Professor Chenyang Lu, Chair

Thermal and power management have become increasingly important for computers. Computing systems from real-time embedded systems to data centers require effective thermal and power management to prevent overheating and save energy. In this dissertation we investigate dynamic thermal and power management for computer systems and buildings. (1) We present thermal control under utilization bound (TCUB), a novel control-theoretic thermal management algorithm designed for single core real-time embedded systems. A salient feature of TCUB is to maintain both desired processor temperature and real-time performance. (2) To address unique challenges posed by multicore processors, we develop the real-time multicore thermal control (RT-MTC) algorithm. RT-MTC employs a feedback control loop to enforce the desired temperature and CPU utilization of the multicore platform via dynamic frequency and voltage scaling. (3) We research dynamic thermal management for real-time services running on server clusters. We develop the control-theoretic thermal balancing (CTB) to dynamically balance temperature of servers via distributing clients' service requests to servers. (4) We propose CloudPowerCap, a power cap management system

for virtualized cloud computing infrastructure. The novelty of CloudPowerCap lies in an integrated approach to coordinate power budget management and resource management in a cloud computing environment.

Chapter 1

Introduction

Thermal and power management have become increasingly important for both computing and physical systems. Fast growing power density and on-chip temperature are key challenges in computer system design. Increased temperatures reduces life span of processors, degrades performance, adversely affects reliability and increases cooling cost and energy. Hence there is widespread interest in thermal management at different levels of computing systems from real-time embedded systems to data centers.

Power management is another critical concern of modern computing systems since it directly affects both operational and deployment cost. For example, a datacenter of 30,000 square feet, which host tens of thousands of servers, consumes 10MW of electricity and requires an accompanying cooling system that costs from 2 to 5 million dollars. Overall, data centers in US consume 100 billion kWh in 2011 according to estimate by the Environmental Protection Agency (EPA) [21]. Moreover, power delivery and cooling limitations in datacenters are bottlenecks of high density configurations to meet the ever increasing performance and scalability demand.

This dissertation focus on studying of thermal and power management of computer systems. The research can be divided into two parts. The first part, from Chapter 2 to Chapter 4, focuses on thermal and power management of real-time systems on different types of computing platforms. Chapter 2 and Chapter 3 investigates thermal management for real-time systems working on single core and multicore processors, respectively. Due to uncertainties in power consumption and workload, we proposed a suite of control-theoretic thermal management approaches to meet both the temperature and real-time performance requirements. Chapter 4 studies thermal management of real-time clusters. In contrast to existing

approach, the proposed approach reduces *hot spots* in the cluster by balancing thermal workload rather than regulating individual servers, improving system throughput without compromising thermal performance.

In the second part, Chapter 5, we focus on power management of distributed systems, specifically, power cap management for cloud computing infrastructure. Our dynamic power cap management, CloudPowerCap, coordinates with existing resource management framework to provide integrated power budget and resource management for virtualized server clusters.

1.1 Feedback Thermal Management for Real-time systems

Real-time embedded systems face significant challenges in thermal management as they adopt modern processors with increasing power density and compact architecture. Such systems must avoid processor overheating while still maintaining desired real-time performance. While modern processors usually rely on hardware throttling mechanisms to prevent overheating, such mechanisms cause performance degradation which is unacceptable for real-time applications. Moreover, real-time systems must deal with a broad range of uncertainties in system characteristics and environmental conditions, such as power consumption variation, ambient temperature fluctuation and thermal fault. Finally, multicore processors induce unique challenges on thermal management due to inter-core thermal coupling and practical constraints of power management mechanism such as Dynamic Voltage and Frequency Scaling (DVFS).

In recent years, control-theoretic thermal management approaches have shown promise in [20, 27, 54, 99, 105, 106] handling uncertainties in thermal characteristics. In contrast to heuristic-based design relying on trial-and-error, control-theoretic approaches provide a scientific framework for systematic design and analysis of thermal control algorithms. The major advantage of adopting feedback control theory in thermal management is to systematically handle uncertainties of thermal dynamics of computing systems.

In this dissertation, we first present *Thermal Control under Utilization Bound (TCUB)*, a novel dynamic thermal management algorithm specifically designed for real-time systems running on single-core processors. TCUB employs feedback control loops to control *both* the processor temperature and CPU utilization by adjusting task rates. In contrast to earlier research on feedback control real-time scheduling that ignores thermal issues [60], TCUB can maintain *both* desired processor temperature and CPU utilization bound, thereby avoiding processor overheating and maintaining desired real-time performance.

Secondly, to address the unique challenges posed by mulitcore processors, we present *Real-Time Multicore Thermal Control (RT-MTC)*, a feedback thermal control algorithm for real-time systems running on multicore processors. RT-MTC employs a feedback control loop that enforces the desired temperature and CPU utilization bounds of embedded real-time systems through DVFS. RT-MTC combines a control-theoretic approach and a practical design to provide a simple, efficient and easily implemented solution to handle challenges and requirements unique to multicore processors.

Finally, we designed *Control-theoretic Thermal Balancing (CTB)*, an feedback thermal management approach for server clusters running real-time services. CTB performs dynamic thermal balancing to reduce the differences among the temperatures of different processors through workload distrbution. Thermal balancing is an attractive approach for thermal management on distributed systems since it can mitigate hot spots without significantly compromising system performance. CTB employs a feedback control loop that periodically monitors the temperature and CPU utilization of different servers in a cluster, and redistributes clients' service requests among different processors to dynamically balance their temperature.

1.2 CloudPowerCap

In many datacenters, server racks are as much as 40 percent underutilized [30]. Rack slots are intentionally left empty to keep the sum of the servers' nameplate power below the power provisioned to the rack. To address rack underutilization, server vendors have introduced support for per-host power caps, which provide a hardware or firmware-enforced limit on the amount of power that the server can draw [19, 39, 43]. However while this approach

improves rack utilization, it burdens the operator with managing the rack power budget across the hosts and, even worse, lacks flexibility to handle workload spikes or to respond to the addition or removal of a rack's powered-on server capacity.

In this dissertation we developed *CloudPowerCap*, a holistic and adaptive solution for power budget management in a virtualized environment. CloudPowerCap manages the power budget for a cluster of virtualized servers, dynamically resetting the per-host power caps for hosts in the cluster. The key of CloudPowerCap is to treat and manage the power cap in close *coordination* with resource management system. CloudPowerCap maps each host's power cap into resources capacity, by which CloudPowerCap can interoperate with a sophisticated resource management system of cloud datacenters, allowing it to manage power caps through the VM resource controls supported by resource management systems. CloudPowerCap provides global fairness on dynamical power caps distribution with robustness for unpredictable workload variation, preventing hosts from gaining unfair entitlement of power caps and enhancing the system's capability to enforce VM placement constraints. To the best of our knowledge, CloudPowerCap is the first holistic framework to provide dynamic power budget management in coordination with a cloud resource management system.

1.3 Contributions

Specifically, this dissertation made the following contributions:

Feedback thermal management for real-time systems on single core processor: We designed *TCUB*, a novel dynamic thermal management algorithm specifically for real-time embedded systems on single processors. TCUB employs feedback control loops to enforce *both* desired processor temperature and CPU utilization by adjusting task rates, thereby avoiding processor overheating and maintaining desired real-time performance.

Feedback thermal management for real-time systems on mulit-core processor: We designed *RT-MTC*, a feedback thermal control algorithm to tackle the challenges posed by multicore processors, which enforces the desired temperature and CPU utilization bounds of real-time embedded systems through realistic DVFS mechanisms.

Feedback thermal balancing for real-time clusters: We designed *CTB*, a control-theoretic thermal balancing approach employing feedback control loop that redistributes clients' service requests among different servers to dynamically balance their temperature.

Dynamic power cap management for cloud computing infrastructure: We designed *CloudPowerCap*, a holistic and adaptive solution for power budget management in a virtualized environment. CloudPowerCap manages the power budget for a cluster of virtualized servers, dynamically resetting the per-host power caps for hosts in the cluster to respect the power budget of the cluster. The key of CloudPowerCap is to treat and manage the power cap in close *coordination* with resource management system.

Chapter 2

Feedback Thermal Management for Real-time Systems on Single Core Processors

2.1 Introduction

Real-time embedded systems face significant challenges in thermal management as they adopt modern processors with increasing power density and compact architecture. Such systems must avoid processor overheating while still maintaining desired real-time performance. While modern processors usually rely on hardware throttling mechanisms to prevent overheating, such mechanisms cause performance degradation unacceptable for real-time applications.

Moreover, real-time embedded systems must deal with a broad range of uncertainties in system characteristics and environmental conditions:

- *Power consumption*: The power consumption of a processor may vary significantly when running different tasks with different instructions [46].
- *Ambient temperature*: In contrast to servers operating in air-conditioned environments, real-time embedded systems may operate in diverse environments under a wide range of ambient temperature.

- *Thermal faults*: Due to their harsh operating conditions embedded systems can be particularly susceptible to failures of cooling subsystems [23].

- *Tasks execution times*: The execution times of many real-time tasks may vary significantly because their executions are strongly influenced by the operating environment and sensor inputs.

To meet these challenges, we present *Thermal Control under Utilization Bound (TCUB)*, a novel dynamic thermal management algorithm specifically designed for real-time embedded systems. TCUB employs feedback control loops to control *both* the processor temperature and CPU utilization by adjusting task rates. In contrast to earlier research on feedback control real-time scheduling that ignores thermal issues [60], TCUB can maintain *both* desired processor temperature and CPU utilization bound, thereby avoiding processor overheating and maintaining desired real-time performance. TCUB has the following salient features.

- TCUB features a nested feedback control structure consisting of (1) a low-rate thermal controller dealing with the slower thermal dynamics, and (2) a high-rate utilization controller handling the faster CPU utilization dynamics caused by uncertainties in task execution times. The thermal controller outputs a set point for the CPU utilization below the schedulable utilization bound of the real-time system. This set point is, in turn, used by the utilization controller to adjust the task rates. The modular control structure allows separate control designs optimized for thermal-protection and utilization-regulation.

- In contrast to earlier research on thermal-ware real-time scheduling that relies in accurate system and task models [14, 42, 92, 93, 102], TCUB is a highly *robust* algorithm that can handle a broad range of uncertainties in terms of processor power consumption, task execution times, thermal faults, and ambient temperature. The robustness of TCUB makes it particularly suitable for real-time embedded systems that operate in highly unpredictable environments.

- TCUB features a simple and efficient thermal controller that integrates a discrete-time-proportional-integral-controller and a traditional anti-windup controller designed to enforce the schedulable utilization bound. The anti-windup controller is necessary

to enforce the schedulable utilization bound that impose hard saturation constraints on the output of the thermal controller (utilization set point). Moreover, the control approach allows rigorous analysis of stability and robustness under uncertainties.

- Extensive simulation results demonstrate the stability and robustness of TCUB under a wide range of uncertainties and operating conditions including varying tasks execution times, power consumption and ambient temperature, as well as thermal faults.

The rest of the chapter is organized as follows. Section 2.3 presents a difference equation model that characterizes the thermal dynamics of real-time systems and the goal of thermal control for real-time systems. Section 2.5 details the design and stability analysis of TCUB. Section 2.6 provides simulation results. Section 2.2 introduces related work. Section 2.7 summarizes this chapter.

2.2 Related Work

Thermal management of real-time systems received significant attention recently. Some researchers explore thermal-aware real-time scheduling [14, 42, 92, 93, 102] to enforce temperature bounds while meeting real-time performance constraints. Existing thermal-aware real-time scheduling algorithms rely on accurate knowledge of the system characteristics such as task execution times and power consumption, which may vary dynamically at run time for real-time systems. In contrast, thanks to its robust feedback control approach, TCUB is especially designed to handle a broad range of uncertainties.

A multitude of feedback real-time scheduling and utilization control algorithms have been proposed in recent years, [4, 9, 47, 80, 83, 94, 108], but they are not cognizant of processor temperature. In contrast, TCUB is designed to control *both* the real-time performance and the processor temperature. While TCUB incorporates a utilization controller, the key contribution of this work is the nested control architecture and the novel thermal controller that can handle the utilization bound constraint needed to enforce desired soft real-time performance.

Several papers [17, 69–71, 99, 104–106] have adopted model predictive control or online convex optimization for dynamic thermal management. None of these works are concerned with maintaining real-time performance or enforce CPU utilization bound. In addition, control approaches based on model predictive control and convex optimization has higher computation complexity than our efficient proportional control approach.

Fu. et al. [27] proposed a model predictive control approach for thermal and utilization control in distributed real-time systems. While it shares similar goals as TCUB, there are several differences between them. First, the algorithm proposed in [27] uses different actuators to control temperature (DVFS) and utilization (task rate adaptation). Instead, TCUB uses a single actuator (task rate adaptation) to control both temperature and utilization. This makes TCUB a more general solution applicable to a broader range of real-time systems including those running on embedded hardware that does not support DVFS. At the same time, relying on a single actuator also poses unique challenges since temperature and utilization control are closely coupled due to the shared actuator. Second, our control design is fundamentally different from the model predictive control approach adopted in [27]. Our novel control design result in a simple and efficient nested control algorithm with $\mathcal{O}(1)$ run-time overhead. In contrast, the model predictive controller [27] relies on a least-square estimator with polynomial complexity to solve the control output over the control and prediction horizon. The simplicity and efficiency of TCUB make it a practical solution even for resource-limited embedded processors. Finally, our control approach allows rigorous robustness analysis. Since our robustness analysis is based on the necessary and sufficient conditions required of the Nyquist stability criteria, we prove and demonstrate how our controller can respond quickly while operating under system uncertainties. In contrast, the small-gain conditions [103] required to satisfy robustness criteria of the proposed model predictive controller presented in [27] tend to be conservative and computationally intensive to verify [107]. Loosening these model uncertainty constraints for model-predictive controllers is a daunting task as noted in [63] and currently being addressed in [32, 50, 59].

2.3 Problem Formulation

In this section we first present the system model adopted in this work, and then discuss the goals of thermal control for real-time systems.

2.3.1 System Models

A key feature of our system model is that it characterizes the *uncertainties* in real-time systems in terms of tasks execution times, power consumption, ambient temperature, and thermal faults. We assume a single CPU real-time system running n independent, periodic real-time tasks, $\{T_i | 1 \leq i \leq n\}$. Each task T_i has a period p_i. The rate r_i of the task T_i is defined as $r_i = \frac{1}{p_i}$. Each task has a soft deadline related to its period and an estimated execution time c_i known at design time. However, the actual execution time a_i at run time is unknown and may deviate from c_i.

The rate r_i of the task T_i can be dynamically adjusted within a range $[R_{\min,i}, R_{\max,i}]$. Earlier works have shown that task rates in many real-time applications (e.g., digital feedback control [12] and multimedia [10]) can be adjusted in certain ranges without causing system failure. A task running at a higher rate contributes a higher value to the application at the cost of higher CPU utilization.

When tasks are running on the processor, the active power consumed by the processor fluctuates significantly. Earlier work refers to such significant power variation during run time as power phase behavior [46]. At the instruction level, different instruction types, inter-instruction overhead, memory system states and pipeline related effects cause power fluctuation [85]. Consequently, while the processor's *estimated* active power, P_a, is known, the actual active power of the processor may deviate from the estimate at run time. When the processor is idle, the processor consumes idle power P_{idle}.

We adopt the well known thermal RC model to characterize the thermal dynamics of the processor [5, 23]:

$$\frac{dT(t)}{dt} = -b_2(T(t) - T_o) + b_1 P(t) \tag{2.1}$$

where $T(t)$ is the temperature of the processor, T_o is ambient temperature, $P(t)$ is the actual power consumed by the processor, $b_1 = \frac{1}{C_{\mathsf{th}}}$ and $b_2 = \frac{1}{R_{\mathsf{th}}C_{\mathsf{th}}}$, where C_{th} is heat capacity and R_{th} is heat resistance. As a high level model thermal RC model is efficient for thermal control design and computation comparing to the architecture level thermal model like Hotspot [41] because of its simple structure. However thermal RC model may introduce model error As embedded systems may operate in diverse environments, the ambient temperature T_0 may change. Moreover, thermal faults (e.g., fan failure) may cause significant change to the thermal resistance [23]. A thermal control algorithm designed for real-time systems must handle these uncertainties at run time.

2.3.2 Design Goals

Our thermal control algorithm is designed to meet two primary requirements: (1) to prevent processor overheating, and (2) to maintain desired soft real-time performance. Due to the uncertainties faced by real-time systems, TCUB adopts a feedback control approach that dynamically controls the processor temperature and real-time performance. It allows users to specify a temperature set point T_R, maximum and minimum CPU utilization bound $U_{\max}$, $U_{\min}$. For processors support hardware throttling, the temperature set point is below the temperature threshold for hardware throttling so as to avoid unpredictable performance degradation caused by throttling. For processors that do not support throttling, the temperature set point should be below the maximum temperature tolerable to the processor. The maximum CPU utilization bound $U_{\max}$ should be below the schedulable CPU utilization bound of the real-time scheduling policy. For example, the schedulable CPU utilization bound of Rate Monotonic Scheduling (RMS) is $U_{\max} = n(2^{\frac{1}{n}} - 1)$, where n is the number of the periodic real-time tasks [56]. The minimum CPU utilization bound $U_{\min}$ can be determined by *minimum allowable tasks rate* $R_{\min,i}$ for a given system.

TCUB is designed to prevent processor overheating by keeping the temperature close to the temperature set point T_R and to maintain desired software real-time performance by enforcing the CPU utilization bound $U_{\max}$.[1] Moreover, TCUB must handle uncertainties in terms of power consumption, task execution times, ambient temperature, and thermal faults.

[1] As TCUB only controls the average CPU utilization dynamically, it is not suitable for hard real-time systems.

Finally, the control algorithm should be simple and efficient to provide a practical solution for resource-limited embedded systems.

2.4 Overview of TCUB

We propose a nested feedback control approach to manage both temperature and CPU utilization. As shown in Fig. 2.1, there are two control loops in TCUB that operate at different time scales. The outer loop is responsible for thermal control and runs at a lower rate than the inner loop responsible for utilization control. In the outer loop, the thermal controller aims to enforce the specified temperature set point T_R. At the end of the k^{th} sampling period of the outer loop, the thermal controller computes the utilization set point $U_s(k)$ for the utilization controller of the inner loop based on the measured temperature $T(k)$ provided by the thermal monitor. The inner-loop utilization controller ensures that the utilization converges to the set point $U_s(k)$ computed by the thermal controller by adjusting the task rates. At the k'^{th} sampling period of the inner loop, the utilization controller outputs the task rate change $\Delta r(k')$ based on the measured utilization $U(k')$. The rate actuator adjusts tasks rate based on the output of the utilization controller. Our multi-rate nested control approach has several important advantages.

1. The thermal dynamics are typically significantly slower than the utilization dynamics, which motivates a multi-rate control approach. The processor thermal-control problem usually involves a *large* thermal time-constant whereas existing utilization controllers (which we incorporate into our design) typically have dynamic responses within a few seconds [60].

2. Unlike the computationally intensive model predictive control adopted by earlier work on thermal control [27], our proposed nested control architecture greatly simplifies the control algorithms. It requires neither complicated gain-scheduling tables nor complicated on-line optimization algorithms. The lower rate of the thermal-control loop further reduces computational overhead.

3. We provide a stability and robustness analysis for the thermal-controller, based on the necessary and sufficient Nyquist Stability criterion which allows us to *directly* relate uncertain physical properties of our thermal-dynamic control problem, whereas the model predictive control approach [27] has to rely on a *conservative* small gain assumption and offers little insight into the physical parameter uncertainties which directly affect stability and performance.

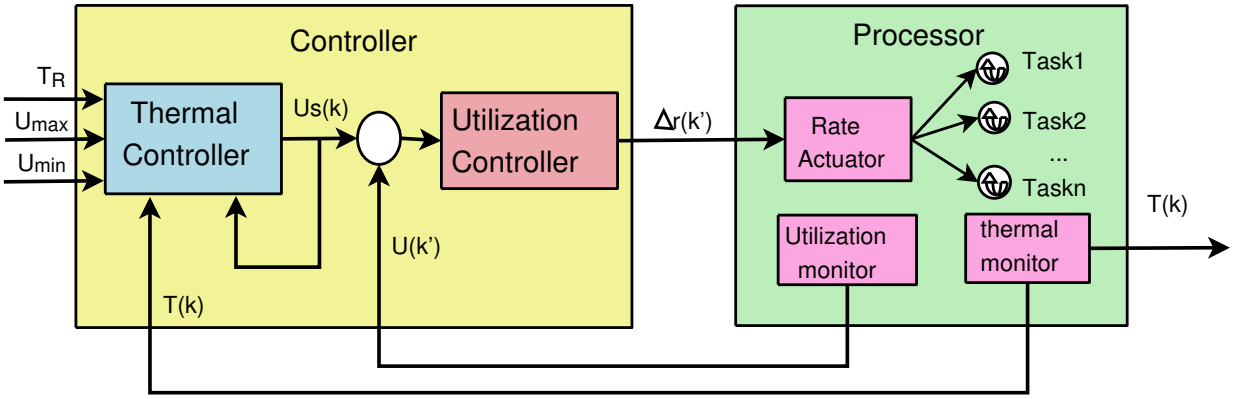


Figure 2.1: TCUB structure

Specifically, the nested control loops work as follows. The thermal and utilization controller employ two sampling periods: T_s, which denotes the sampling period of the processor's temperature; and T_u, which is the sampling period of the utilization ($T_u < T_s$). At the end of the k^{th} temperature sampling period, the feedback loop is invoked and executes the following steps:

1. The temperature monitor sends the processor's temperature $T(k)$ to the thermal controller over the last sampling period.

2. The thermal controller calculates the utilization *set point* of the processor, $U_s(k)$, based on $T(k)$ and temperature reference. It then sends $U_s(k)$ to the utilization controller. Note $U_s(k)$ is effectively held for m samples in which m is a positive integer which relates the outer-loop sample time T_s to the inner-loop sample time T_u such that $T_s = mT_u$.

3. The utilization controller adjusts the task rates through a rate actuator in each T_u sampling period so as to track the utilization set point $U_s(k)$. TCUB employs FC-U [60] as the utilization controller, which uses a Proportional controller to track the utilization set point.

A benefit of our nested control structure is modular design, that is, we can design the two control loops separately. For utilization control loop we reuse the well studied feedback control utilization controller FC-U [60]. FC-U periodically monitors the utilization of the CPU, computes the control output based on difference between current utilization and the utilization setpoint and then calls rate adaptor to change utilization according to control output. FC-U is implemented in FCS/nORB, a real-time middleware on a Linux platform [97]. Since the effectiveness of FC-U is justified by experiments and implementation, in the following sections, we focus on the thermal controller design and stability analysis.

2.5 Thermal Control Design and Analysis

In this section we describe the control design and analysis of TCUB. In the following sections we present the design of thermal controller and the stability analysis.

2.5.1 Dynamic Model for Thermal Control

As a foundation for the design of the thermal controller, we derive a discrete-time, difference equation model that characterizes the dynamic relationship between the CPU utilization $U(k)$ (the control input) and the processor temperature $T(k)$ (the controlled variable). We first characterize the relationship between the power consumption and the CPU utilization and then derive a discrete-time model based on the thermal RC model .

First, we characterize the relationship between the power consumption of the processor and its CPU utilization. Let $U(k)$ denote the CPU utilization in the k^{th} sampling period. The average power of the processor in k^{th} sampling period, $\bar{P}(k)$, has the following relationship

with $U(k)$:

$$\bar{P}(k) = G_p P_a U(k) + P_{\text{idle}}(1 - U(k)) = (G_p P_a - P_{\text{idle}})U(k) + P_{\text{idle}} \tag{2.2}$$

where G_p represents the ratio between the actual active power at run time and the estimated active power P_a. In (2.2), $G_p P_a$ is the actual active power, and $U(k)$ is the fraction of time when the CPU is active. P_{idle} is the power when the CPU is idle, and $1-U(k)$ is the fraction time when the CPU is idle. The same power model is also used in temperature simulation of server systems [37].

Next, we transform the thermal RC model (2.1) to a discrete-time model. Denote the Laplace transform of the difference between processor's temperature and ambient temperature, $T(t)-T_o$, as $T(s)$ as well as $P(t)$ as $P(s)$ from (2.1), we have the following model

$$T(s) = \frac{R_{\mathsf{th}}}{R_{\mathsf{th}} C_{\mathsf{th}} s + 1} P(s) + \frac{1}{R_{\mathsf{th}} C_{\mathsf{th}} s + 1} T_o. \tag{2.3}$$

For control analysis we need to derive a discrete-time model to *approximate* this system. The thermal controller issues a fixed-periodic utilization set point which the inner-loop utilization controller closely and quickly regulates to. This utilization set point is proportional to the average power consumed by the processor. As previously mentioned, the thermal-time constant is large, therefore the effects of transients are *negligible*. Thus a ZOH-equivalent model is appropriate to approximate a discrete-time model of the thermal dynamics of the system. It is straightforward to derive the linear ZOH-equivalent discrete time model from (2.3) as follows [25] :

$$T(k+1) = \Phi T(k) + (1 - \Phi)T_o + R_{\mathsf{th}}(1 - \Phi)P(k) \tag{2.4}$$

where k represents k^{th} sampling period, $\Phi = \exp(-\frac{T_s}{R_{\mathsf{th}} C_{\mathsf{th}}})$ and T_s is the sampling period.

Then we combine the thermal RC model (2.4) and the relationship between power and utilization (2.2), specifically, by substituting $P(k)$ for $\bar{P}(k)$, we could derive the model employed in thermal control:

$$T(k+1) = \Phi T(k) + R_{\mathsf{th}}(1 - \Phi)(G_a P_a - P_{\text{idle}})U(k) + R_{\mathsf{th}}(1 - \Phi)P_{\text{idle}} + (1 - \Phi)T_o. \tag{2.5}$$

2.5.2 Thermal Controller Design

The principal challenge for the thermal controller design is to ensure that a maximum allowable temperature T_R is not exceeded while the thermal-control output $U_s(k)$ is subject to actuator saturation which is governed by utilization bound $U_{\max}(0 \leq U_{\max} \leq 1)$. The thermal controller is required to regulate the temperature of the processor to subject to the utilization constraint given its output $U_s(k)$. A proportional-integrator (PI) controller with an integrator-anti-windup controller is proposed to determine $U_s(k)$ while addressing actuator limitations in order to guarantee stability. This simple yet elegant outer-thermal control loop can be run at a significantly lower-rate without any noticeable performance loss due to the systems large thermal time constant.

The structure of thermal controller we proposed is illustrated in Fig. 2.2. It consists of a PI controller (denoted as $K(z)$), an anti-windup controller (denoted as $\hat{H}(z)$) which is determined from a processor's thermal model $\hat{H}(z)$ and a saturation block. The PI controller's output is limited by the saturated block and then the utilization set point output by the thermal controller cannot surpass the utilization bound assigned by the users. Essentially anti-windup controller transforms nonlinear behavior of the real-time systems induced by the utilization bounds to linear behavior so that normal linear control design could be exploited. The input of the PI controller is the error between the reference trajectory and linearized

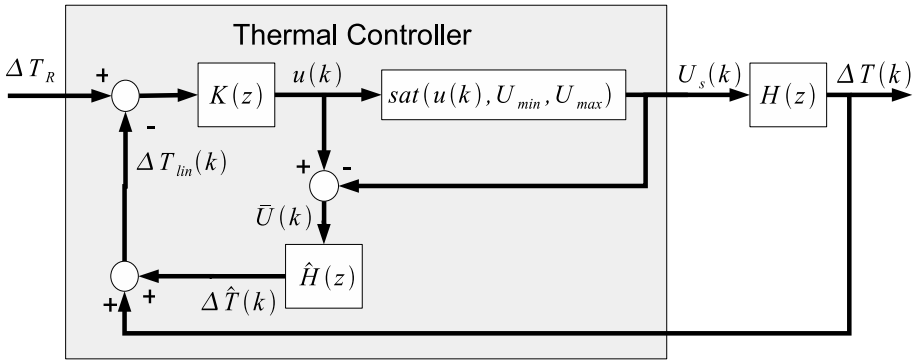


Figure 2.2: Proposed thermal control structure.

temperature $\Delta T_{\text{lin}}(k)$. The control output of the PI controller, $u(k)$, is limited to enforce

utilization bounds by the saturated block, $U_s(k) = \text{sat}(u(k), U_{\min}, U_{\max})$, in which

$$\text{sat}(x, x_{\min}, x_{\max}) = \begin{cases} x_{\min}, \text{ if } x < x_{\min} \\ x_{\max}, \text{ if } x > x_{\max} \\ x, \text{ otherwise.} \end{cases}$$

The error between $U_s(k)$ and $u(k)$, denoted as $\bar{U}(k)$, is passed through a thermal model of the processor (denoted $\hat{H}(z)$) which generates a compensation term $\Delta\hat{T}(k)$, when combined with the actual processor temperature difference $\Delta T(k)$, a *linearized* temperature difference ($\Delta T_{\text{lin}}(k) = \Delta\hat{T}(k) + \Delta T(k)$) is fed-back to the controller $K(z)$ in order to guarantee stability. This compensation is also known as *anti-windup* control. It is noted that the thermal model of the processor is used here without considering dynamic of the utilization controller. This is one of the benefits of nested control structure, that is, we can design the thermal and utilization controller separately. In order to describe our implementation of the thermal controller, as presented in Algorithm 1, we denote $\hat{T}_{idle}$ as an estimate of the idle temperature $T_{idle}(t)$ and $\hat{T}_o$ as either an estimate or measurement of ambient temperature T_o.

For thermal controller design, we rewrite the model (2.5) in a more compact form. Note that the temperature $T(k)$ depends ultimately on the ambient temperature T_o, the idle temperature component T_{idle} which depends on the idle power component P_{idle} such that $T_{\text{idle}} = R_{\mathsf{th}} P_{\text{idle}}$, and the active power component $\Delta T(k)$, that is, $T(t) = \Delta T(t) + T_o + T_{\text{idle}}$. Then the model (2.5) could be rewritten as

$$\Delta T(k+1) = \Phi \Delta T(k) + \Gamma U(k) \tag{2.6}$$

where $\Gamma = k_p R_{\mathsf{th}}(1 - \Phi)$ and $k_p = (G_a P_a - P_{\text{idle}})$. In model (2.6) uncertainty in G_p can be expressed in terms of the following bounds on the *actual power gain* k_p such that $k_{p\min} \leq k_p \leq k_{p\max}$.

In z-domain the model (2.6) can be written as follows

$$H(z) = \frac{\Delta T(z)}{U(z)} = \frac{\Gamma}{z - \Phi}. \tag{2.7}$$

To design the thermal controller with the proposed structure we follow two steps. First a nominal linear controller $K(z)$ ignoring the saturating limit is designed. In this work the nominal linear controller is a PI-controller, $K(s) = K_{\mathrm{P}} + K_{\mathrm{I}}\frac{s+\omega_{\mathrm{I}}}{s}$. The discrete time controller $K(z)$ is synthesized using the IPESH-transform from the continuous time controller model $K(s)$. The IPESH-transform, like the bilinear-transform, is both a passivity and stability preserving transform which can be applied to any linear-time invariant model $K(s)$ except that it will not *suffer from warping effects* and therefore closely matches the magnitude response up to the Nyquist frequency $\frac{\pi}{T_s}$ [48, 49]. The result discrete time controller is:

$$K(z) = K_{\mathrm{P}} + K_{\mathrm{I}}\left(1 + \frac{\omega_{\mathrm{I}}T_s}{2}\right)\frac{z - \frac{2-\omega_{\mathrm{I}}T_s}{2+\omega_{\mathrm{I}}T_s}}{z-1}.$$

Secondly, an anti-windup controller $\hat{H}(z)$ is designed to limit performance deterioration in the event of a control constraints being encountered. The details of anti-windup controller are presented in Section 2.5.3.

Algorithm 1 describes workflow of the thermal controller and the derivation of thermal controller related parameters used in the algorithm are explained in Section 2.5.3.

2.5.3 Stability Analysis

The section analyzes the stability of the proposed control framework. For a real-time system under thermal control, stability ensures that the processor temperature converges to the temperature set-point. In order to discuss stability, we recall the following definition and the Nyquist stability theorem.

Definition 1. *A* stable *discrete-time linear time invariant (LTI) system is one in which all poles are inside the unit circle.*

For our control structure, it should be intuitive from viewing Fig. 2.2 that there are only two cases to maintain stability. The first case, when the control input $U_{\min} \leq u(k) \leq U_{\max}$ (which implies that $\bar{U}(k) = 0$) we want to enforce stability of the *active* closed-loop system consisting of $K(z)$ and $H(z)$, and stability of $\hat{H}(z)$. For the second case, when the control

Algorithm 1 Thermal Controller

$T(k)$: temperature
$U_s(k)$: thermal-control Output
T_R: temperature set point
$\hat{T}_0, \hat{T}_{\text{idle}}$: estimated environment and idle temperature
$U_{\max}$, $U_{\min}$: utilization bounds
$K_p, K_I, \omega, T_s, \hat{\Phi}, \hat{\Gamma}$: controller parameters

Compute temperature difference set point $\Delta T_R(k) = T_R - \left(\hat{T}_0 + \hat{T}_{\text{idle}}\right)$ ▷ At the end of the k^{th} sampling period
The linearized temperature is computed by $\Delta T_{\text{lin}}(k) = T(k) - \left(\hat{T}_0 + \hat{T}_{\text{idle}}\right) + \Delta\hat{T}(k)$
$e(k) = (\Delta T_R(k) - \Delta T_{\text{lin}}(k))$ ▷ PI controller
$u(k) = u(k-1) + K_P(e(k) - e(k-1)) + K_I\left(1 + \frac{\omega_I T_s}{2}\right)\left(e(k) - \frac{2-\omega_I T_s}{2+\omega_I T_s}e(k-1)\right)$
if $U_{\min} \leq u(k) \leq U_{\max}$ **then**
    $U_s(k) = u(k)$
else
    if $u(k) < U_{\min}$ **then**
        $U_s(k) = U_{\min}$
    else
        $U_s(k) = U_{\max}$
    end if
end if
$\bar{U}(k) = u(k) - U_s(k)$
$\Delta\hat{T}(k+1) = \hat{\Phi}\Delta\hat{T}(k) + \hat{\Gamma}\bar{U}(k)$

input saturates $u(k) < U_{\min}$ or $u(k) > U_{\max}$, we want to enforce stability of the *active* closed-loop system consisting of $K(z)$ and $\hat{H}(z)$, and stability of $H(z)$.

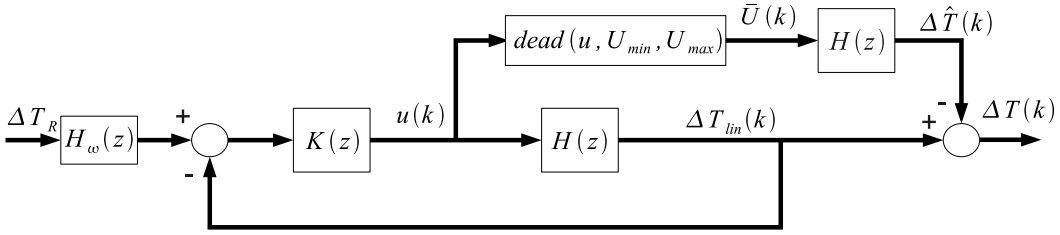


Figure 2.3: Resulting feedback control structure when $H(z) = \hat{H}(z)$.

For the first case, as is assumed in [33,38], stability of this system will first be considered for the special case in which $\hat{H}(z) = H(z)$. In such case it is straightforward to show that Fig. 2.2 can be drawn in the equivalent form as depicted in Fig. 2.3. The function $\text{dead}(u, U_{\min}, U_{\max})$ is implemented as follows:

$$\text{dead}(u, U_{\min}, U_{\max}) = \begin{cases} (u - U_{\min}), \text{ if } u \leq U_{\min} \\ 0, \text{ if } U_{\min} < u < U_{\max} \\ (u - U_{\max}), \text{ otherwise.} \end{cases}$$

The verification of stability of the closed-loop system in Fig. 2.3 is based on the well-know Nyquist stability criteria in frequency domain [75], from which and Fig. 2.3 we obtain Theorem 1 to verify stability of the our proposed control structure (Fig. 2.2) directly.

Theorem 1. *The closed-loop system depicted in Fig. 2.2 is stable if 1) $K(z)H(z)$ satisfies Nyquist stability criteria; 2) $\hat{H}(z) = H(z)$. In addition, if the output $\Delta T(k)$ is to reach a steady-state output for a given input ΔT_R, then $\hat{H}(z)$ should be stable.*

For the second case, to avoid introducing additional terms and complexity, we simply note that when $\hat{H}(z) = H(z)(1+\Delta(z))$, Fig. 2.2 can be shown to be in the equivalent form depicted in Fig. 2.4. Therefore, when checking for stability, one should verify whether $K(z)\hat{H}(z)$ also satisfy the Nyquist stability criteria.

Nyquist stability criteria and Lemma 1 lead us to the following theorem:

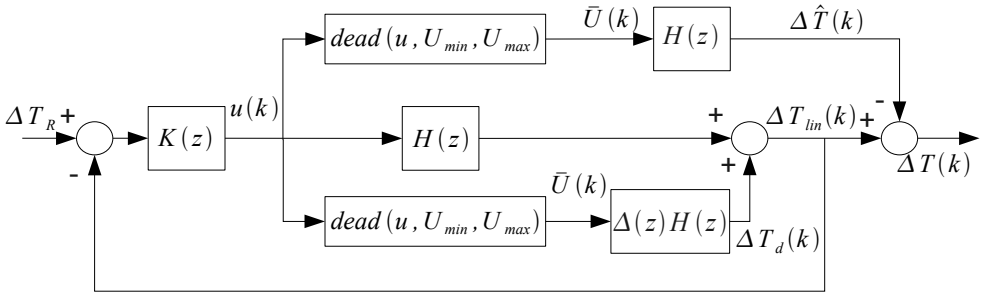


Figure 2.4: Equivalent control structure given that $\hat{H}(z) = (1 + \Delta(z))H(z)$.

Theorem 2. *The closed-loop system with controller*

$$K(z) = K_P + K_I\left(1 + \frac{\omega_I T_s}{2}\right)\frac{z - \frac{2-\omega_I T_s}{2+\omega_I T_s}}{z-1}$$

depicted in Fig. 2.2 in which ΔT_R is the input, and ΔT_{lin} is the output is stable if:

1. $\hat{H}(z) = \frac{\hat{\Gamma}}{z-\hat{\Phi}}$, $\hat{\Gamma} \leq \Gamma_{\max}$, $\hat{\Phi} \leq \Phi_{\max}$
2. $K_P = K_I = k_{GM}\frac{1+\Phi_{\max}}{2\Gamma_{\max}}$

in which $k_{GM} = 10^{-\frac{GM}{20}}$, $\Phi_{\max} = \exp(-\frac{T_s}{R_{\text{th}\max}C_{\text{th}}})$, $\Gamma_{\max} = k_{p\max}R_{\text{th}\max}(1-\Phi_{\max})$ and $\omega_I = \frac{2(1-\Phi_{\max})}{T_s(1+\Phi_{\max})}$. where GM is the desired worst-case gain margin and , $0 \leq GM < \infty$.

Proof. We show a brief proof of Theorem 2. Let us first consider the case of the closed loop only with $K(z)$ and $H(z)$. The plant-controller loop-product can now be written in the following form:

$$K(z)H(z) = \frac{K_{\text{P}}\Gamma}{z-\Phi} + K_{\text{I}}\left(1 + \frac{\omega_{\text{I}}T_s}{2}\right)\frac{z-\Phi_{\max}}{z-\Phi}\frac{\Gamma}{z-1}. \tag{2.8}$$

The models of (2.8) and (2.5) indicate that no poles exist outside the unit circle for all $T_s < \infty$; therefore, Lemma 1 will always be satisfied if

$$|K(e^{j\pi})H(e^{j\pi})| \leq 1, \text{ and } \Phi_{\max} = \exp(-\frac{T_s}{R_{\text{th}\max}C_{\text{th}}}) \geq \Phi.$$

These two conditions are sufficient that the phase margin will be greater than zero when $\omega = \pi$. In particular we note that if we assume that $\omega_{\mathrm{I}} = \frac{2(1-\Phi_{\max})}{T_s(1+\Phi_{\max})}$ then by cross multiplication $\Phi_{\max} = \frac{2-\omega_{\mathrm{I}}T_s}{2+\omega_{\mathrm{I}}T_s}$. Therefore, our proposed controller has the following form

$$K(z) = K_{\mathrm{P}} + K_{\mathrm{I}}\frac{2}{1+\Phi_{\max}}\left(\frac{z-\Phi_{\max}}{z-1}\right)$$

so that

$$K(z)H(z) = \frac{K_{\mathrm{P}}\Gamma}{z-\Phi} + \frac{2K_{\mathrm{I}}}{1+\Phi_{\max}}\frac{(z-\Phi_{\max})\Gamma}{(z-\Phi)(z-1)} = \left(\frac{\Gamma}{1+\Phi_{\max}}\right)\frac{z-\frac{K_{\mathrm{P}}(1+\Phi_{\max})+\Phi_{\max}2K_{\mathrm{I}}}{K_{\mathrm{P}}(1+\Phi_{\max})+2K_{\mathrm{I}}}}{(z-1)(z-\Phi)} \tag{2.9}$$

from the corresponding pole-zero plot, it is evident that the magnitude $|K(e^{j\omega})H(e^{j\omega})|$ is a smoothly decreasing function in which the phase $\angle K(e^{j\omega})H(e^{j\omega}) > -\pi$ for $\omega \in [0,\pi)$ if

$$\Phi < \frac{K_{\mathrm{P}}(1+\Phi_{\max})+\Phi_{\max}2K_{\mathrm{I}}}{K_{\mathrm{P}}(1+\Phi_{\max})+2K_{\mathrm{I}}} < 1 \text{ holds.}$$

Indeed, the above inequality will be shown to hold if $\Phi_{\max} > \Phi$. It is therefore sufficient to let the magnitude of $|K(e^{j\pi})H(e^{j\pi})| < 1$ or the magnitude of the respective proportional term (involving K_{P}) and integral term (involving K_{I}) to each be less than one-half when $\omega = \pi$ and can indeed be readily verified from our first expression given for $K(z)H(z)$, and carefully noting the relationship between the ratio involving Φ and Γ in which

$$\frac{K_{\mathrm{P}}}{k_{\mathrm{GM}}} < \frac{|e^{j\pi}-\Phi|}{2\Gamma} \leq \frac{1+\Phi_{\max}}{2\Gamma_{\max}}, \frac{K_{\mathrm{I}}}{k_{\mathrm{GM}}} < \frac{1+\Phi_{\max}}{4\Gamma}\frac{|(e^{j\pi}-\Phi)(e^{j\pi}-1)|}{|e^{j\pi}-\Phi_{\max}|} \leq \frac{1+\Phi_{\max}}{2\Gamma_{\max}}.$$

We will always know what $U_{\max}$ will be as it is dictated by the scheduler chosen, however, some uncertainty may remain on choosing the lower-limit $U_{\min}$ due to task execution time. Therefore even choosing the ultimate lower-bound $U_{\min} = 0$ can always be a safe choice even if $U_{\min} > 0$ in that it will result in a slight sub-optimal lag in allowing the controller to increase the utilization levels due to a decrease in environmental temperature for example. Considering that environmental temperature changes are fairly slow, this slight lag is typically unnoticeable. For a more detailed discussion on anti-windup control, we refer the reader to [33, 38]. □

The Theorem 2 reveals the appealing feature of our thermal controller, that is, its robustness under power change and thermal fault can be guaranteed analytically. Since k_p involves uncertainty of power change represented by G_p according its definition, $k_p = (G_p P_a - P_{\text{idle}})$, $k_{p\,\max}$ corresponds to the maximum actual power changes that TCUB can cancel. For example, if $k_{p\,\max} = 510$, $P_a = 51.9W$ and $P_{\text{idle}} = 13.3W$, we can calculate that the upper limit of G_p is 10.11, that is, even if the actual power is 10.11 times by the estimated power, the thermal controller still can stabilize the system. Similarly, the capability of TCUB to handle thermal fault (modeled by increased thermal resistance) is represented by $R_{\text{th}\,\max}$. Note that the Theorem 2 not only provides robustness guarantee but also design of the anti-windup controller.

Moreover, another appealing property is isolation of varied ambient temperature in our thermal controller. This property is provided by Lemma 1.

Lemma 1. *If the processor's temperature converges, it converges to the temperature set point even the estimated ambient temperature, $\hat{T}_o$, and idle temperature $\hat{T}_{\text{idle}}$ are employed.*

Proof. The proof is straightforward. it is obvious that for the *steady-state case* when the $u(k) = U_s(k)$ that $\Delta T_R(k) = \Delta T_{\text{lin}}(k) = \Delta T_{\text{fb}}(k)$ due to the integrator term in $K(z)$. Therefore, from the following equation, $\Delta T_R = T_R - (\hat{T}_o + \hat{T}_{\text{idle}}) = T(k) - (\hat{T}_o + \hat{T}_{\text{idle}}) = \Delta T_{\text{fb}}(k)$, we have $T_R = T(k)$, that is, the processor's temperature converges to the temperature set point. □

It is noted that due to the minimum task rate constraints, there exists a lower bound for the feasible utilization, which in turn results in a lower bound for the feasible temperature. The lower bounds for the utilization and temperature are related to the rate constraints, the actual execution times, and the actual power consumption. TCUB can achieve satisfactory thermal and real-time performance only if both the given temperature set-point and the utilization bound are feasible under the task rate constraints.

2.5.4 Sensitivity Analysis

In the preceding analysis we provided necessary and sufficient conditions for stability assuming that environmental temperature T_o and idle power P_{idle} (corresponding idle temperature T_{idle}) remained constant. For simplicity of discussion, we will: i) further assume that idle power is constant and equal to zero[2]; ii) we will treat environmental temperature $T_o(k) \neq T_o(k+1)$ as a disturbance with the respective z-transform $T_o(z)$; iii) assume that saturation does not occur; iv) $\hat{T}_o = 0$, therefore $\Delta T_R(k) = T_R(k)$. Since $T_{\text{idle}} = 0$ then $\Delta T(k) = T(k) - T_o(k)$ as a result it is a straight forward exercise to show from (2.5) and (2.7) that:

$$\Delta T(z) = \frac{\Gamma}{z-\Phi}U(z) - \frac{z-1}{z-\Phi}T_o(z), T(z) = \frac{\Gamma}{z-\Phi}U(z) + \frac{1-\Phi}{z-\Phi}T_o(z). \tag{2.10}$$

Using (2.5.4) we state Lemma 2.

Lemma 2. *For the control system depicted in Fig. 2.1 in which* $u(k) = U_s(k)$, $T_R(z) = 0$ *the sensitivity transfer function* $S(z)$ *is* $S(z) = \frac{T(z)}{T_o(z)} = \frac{1-\Phi}{z-\Phi+\Gamma K(z)}$. *Furthermore stability is unaffected when* $T_o(k+1) \neq T_o(k)$ *and* $P_{\text{idle}}(k+1) \neq P_{\text{idle}}(k)$.

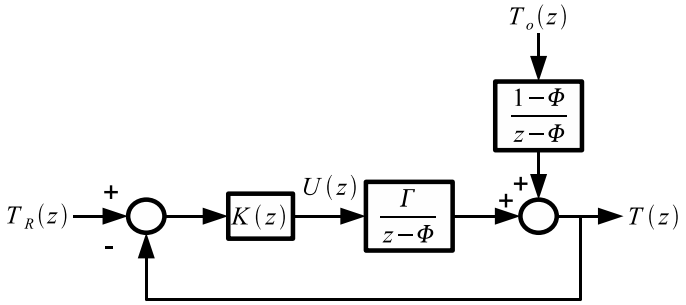


Figure 2.5: Closed-Loop structure when studying sensitivity.

Proof. Fig. 2.5 depicts (2.5.4) and our proposed control structure when saturation is not present. Therefore, using standard closed loop realizations we can show that

$$T(z) = \frac{\Gamma K(z)}{z-\Phi+\Gamma K(z)}T_R(z) + \frac{1-\Phi}{z-\Phi+\Gamma K(z)}T_o(z)$$

[2]Constant idle power has no effect on the sensitive function shown in Lemma 2.

then setting $T_R(z) = 0$ results in the standard sensitivity transfer function $S(z) = \frac{1-\Phi}{z-\Phi+\Gamma K(z)}$. Note also that the sensitivity function has the same characteristic equation as the closed-loop system ($\frac{\Gamma K(z)}{z-\Phi+\Gamma K(z)}$) studied for closed-loop stability. Therefore it is obvious that dynamic effects related to environmental temperature T_o have no effect on stability. An analogous observation can be made if idle power is considered to be time-varying. □

For an example system with parameters from Tab. 2.1 and Tab. 2.2, Fig. 2.6 plots the magnitude of the sensitivity transfer function of the system with respect to frequency. Note that in the whole range of frequency the magnitude is less than 0, which means the effect of T_o converges to 0 in steady state and can not affect the temperature of closed loop system.

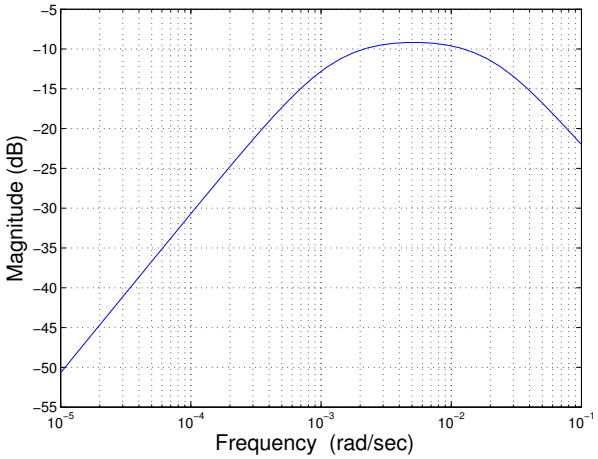


Figure 2.6: Magnitude of sensitivity transfer function of the example system.

2.6 Evaluation

The simulation environment consists of two components: an event driven simulator implemented in C++ and a Simulink© model implemented in MATLAB (R2008a). The simulator simulates a single processor real-time system controlled by TCUB and implements a utilization monitor, a rate actuator and a utilization controller. The Simulink© component implements the thermal controller and models thermal dynamics of the processor. The simulator and the Simulink© component communicate with each other through a TCP connection.

In our simulation, the tasks set running on the processor consists of 10 periodic soft real-time tasks. The Rate Monotonic (RM) scheduling algorithm [56] is employed to schedule all tasks. Initially, the period of each task T_i is randomly generated in the range $[100ms, 200ms]$. Based on the initial tasks rate, the execution times of tasks are deliberately chosen in such a way that the CPU utilization of each task are almost equal and the CPU utilizaiton of all tasks is lower than schedulable CPU utilization bound. The minimum rate of one task equals its execution time while the maximum rate equals 10 times of initial tasks rate. The deadline of each task equals its period.

The processor simulated in our work is a $2.6GHz$ Pentium 4 (P4) processor with $130nm$ Northwood core. All thermal related parameters except thermal capacitance shown in Table 2.1 are based on Intel technical specification [44]. The thermal capacitance is acquired based on the parameters used for simulating P4 on Hotspot [41] , a widely used architecture level simulator, .

Parameter	Notation	Value
Ambient temperature	T_o	$45°C$
Max case temperature	T_c	$75°C$
Estimated Active power	P_a	$51.9W$
Idle power*	P_i	$13.3W$
Thermal Capacitance	C_{th}	$295.7J/K$
Thermal Resistance	R_{th}	$0.467K/W$
Thermal Fault Resistance	R'_{th}	$2R_{\mathsf{th}}$

* Enhanced Halt Mode is available [86]

Table 2.1: Power and thermal parameters of simulated processor.

In the following simulations, we choose $70°C$ as the set point of the processor's temperature. The set point is lower than the maximum case temperature $75°C$ to avoid activation of interal hardware regualtion which ends up unpredictable performance degradation. The thermal fault resistance, R'_{th}, is based on the data reported in [23].

Table 2.2 shows the controller parameters of TCUB which are calculated using the methods discussed in Section 2.5.

K_p	K_i	ω_i	$k_{p\max}$	$R_{\mathsf{th}\max}$	$U_{\max}$	$U_{\min}$	T_R	T_s	K_p	T_u
0.0523	0.0523	0.0036	510	0.934	0.67	0.07	70 $^\circ C$	10s	0.37	1s

Table 2.2: TCUB controller parameters

We compare TCUB against three baseline algorithms[3], OPEN, TC and FC-U. OPEN has no feedback thermal and utilization control loop and statically set task rates based on the *estimated* execution times to achieve the schedulable utilization bound. OPEN represents a static approach commonly used in practice. TC has the same thermal controller as TCUB, but does not include the utilization controller. After the thermal controller outputs the utilization set point, it sets the task rates based on the *estimated* execution times. FC-U [60] is the same utilization control algorithm used in TCUB, but does not has the thermal controller to manage temperature. As subsets of TCUB, TC and FC-U allow us to evaluate the effectiveness of the *integrated* control approach of TCUB for both temperature and utilization.

2.6.1 Power Deviation

This set of simulations is designed to evaluate TCUB when the processor's active power deviates from the estimate, which represents *power phase change* observed in previous empirical studies [46]. We use different *power ratios*, i.e., the ratio between the actual active power to the estimate, in different runs. In the first run the power ratio is 2, i.e., the actual active power is twice the estimate; in the second run, the power ratio is 0.5, i.e., the actual power is half of the estimate. The task execution times are the same as their estimate in this set of experiments.

Fig. 2.7 shows the results when power ratio is 2. As shown in Fig. 2.7(a), the temperature of the processor under TCUB converges to the temperature set point $70^\circ C$ while its utilization remains below the utilization bound. Note that TCUB forces the CPU utilization to remain *lower* than its utilization bound , which is needed in order to maintain the temperature set

[3]While several thermal-aware real-time scheduling algorithms exist in the literature [13, 42, 92, 93], they rely on Dynamic Voltage and Frequency Scaling (DVFS) which is not required by TCUB. The only existing feedback control algorithm for thermal control [27] also require on DVFS and hence will not provide a fair comparison with TCUB. We discuss the related works in detail in Section 2.2.

point due to the high active processor power when the power ratio is 2. In contrast, FC-U, shown in Figure 2.7(d), reaches the utilization bound but it *violates* the temperature set-point. OPEN behaves similarly to FC-U except its achieves slightly higher utilization and temperature because the task rates are configured for the schedulable utilization bound which is higher than the utilization set point of FC-U. TC performs similarly to TCUB. Because the execution times are the same as their estimate in this experiment, the utilization controller is not necessary. There is no deadline miss for all algorithms in this experiment. Note that slight fluctuation of utilization in Fig. 2.7 casued by randomness of execution time of the task set.

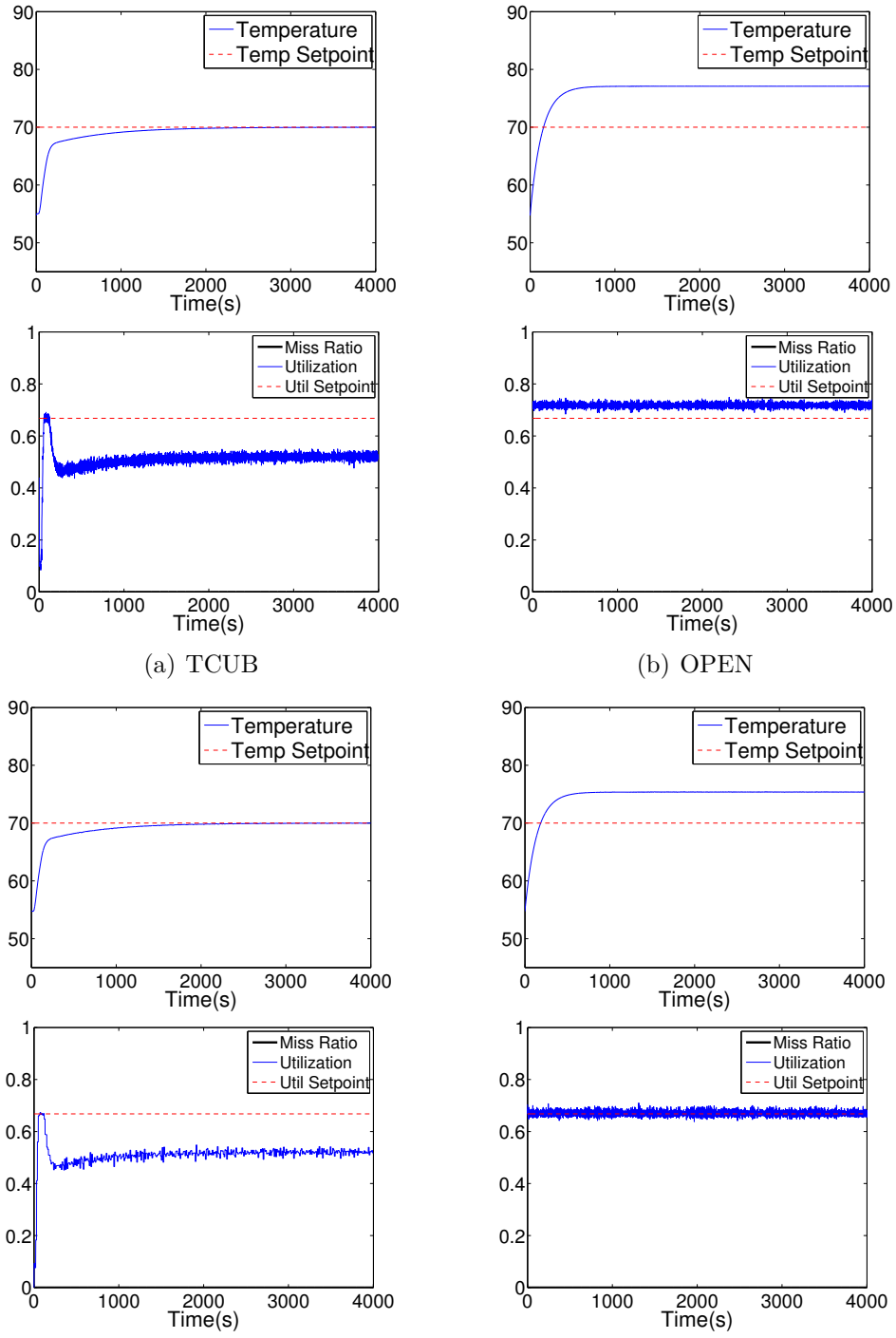
Fig. 2.8 illustrates the simulation results when power ratio is 0.5. TCUB undershoot the temperature set point while the utilization bound is hit in this experiment. Due to the low processor's active power, the utilization bound constraint is activated before the temperature reaches the set-point. As a result, TCUB stops increasing the utilization to enforce the utilization bound. TC behaves similarly to TCUB because the task execution times conform to the estimate. FC-U enforces the utilization bound, which results in a temperature lower than the set-point. OPEN behaves similarly to FC-U.

In summary, this set of experiments demonstrate our thermal controller can effectively handle uncertainties in power consumption, including the cases that either the temperature set point or the utilization bound dominates the thermal control.

Since OPEN has no thermal and utilization control and statically set tasks rates, it can not accommodate variation of system parameters as shown in this experiment and [60]. So in following experiment the results of OPEN are not presented.

2.6.2 Execution Time Variation

This set of experiments is designed to evaluate TCUB under uncertainties in task execution times. We use *execution-time factor (etf)* to denote the ratio between the actual and the estimated execution times. For example, when $etf = 2$, the actual execution time is twice the estimate. We simulate two cases with $etf = 2$ and $etf = 0.5$ in this set experiment



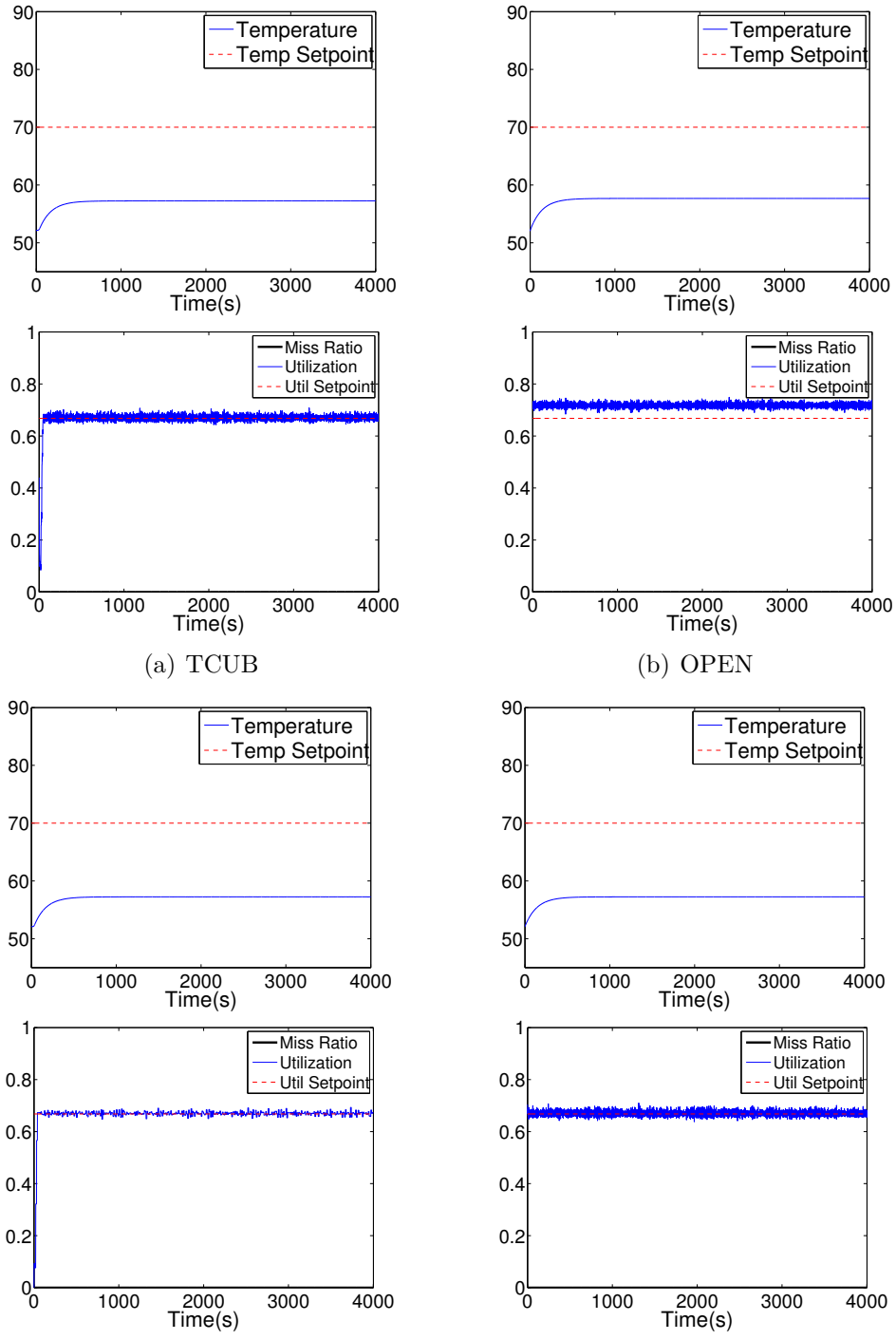
(a) TCUB

(b) OPEN

(c) TC

(d) FC-U

Figure 2.7: Performance comparison when power ratio is 2.



(a) TCUB

(b) OPEN

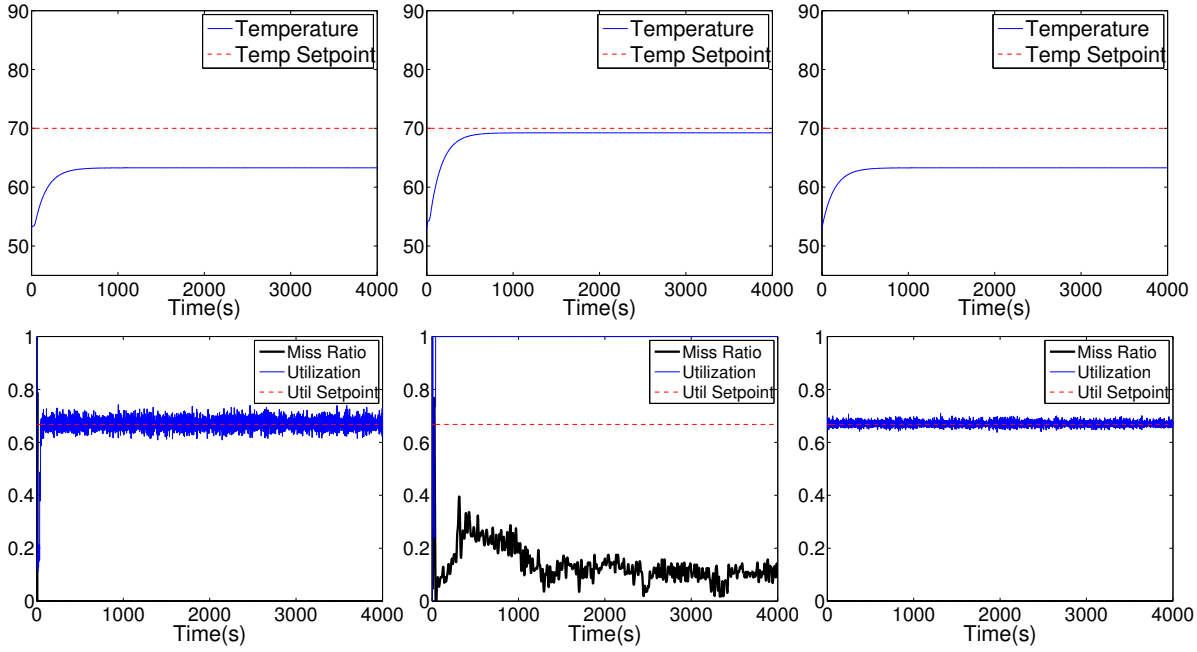
(c) TC

(d) FC-U

Figure 2.8: Performance comparison when power ratio is 0.5.

respectively. Meanwhile, in this experiment we set the power ratio as 1, i.e., the processor's active powers is the same as the estimate.

The results with $etf = 2$ are shown in Fig. 2.9. Although the actual execution times of tasks exceed the estimate by 100%, TCUB successfully enforces the utilization bound. Since the utilization bound is activated first, the processor's temperature is still below the set point. No deadline miss is observed under TCUB. This result demonstrates that TCUB effectively handles uncertainties in task execution times through the utilization controller. Similarly, FC-U enforces the utilization bound and achieves the temperature lower than the set point also. In contrast, TC causes 100% CPU utilization and a significant number of deadline misses since it adjusts task rates based on their *estimated* execution times.

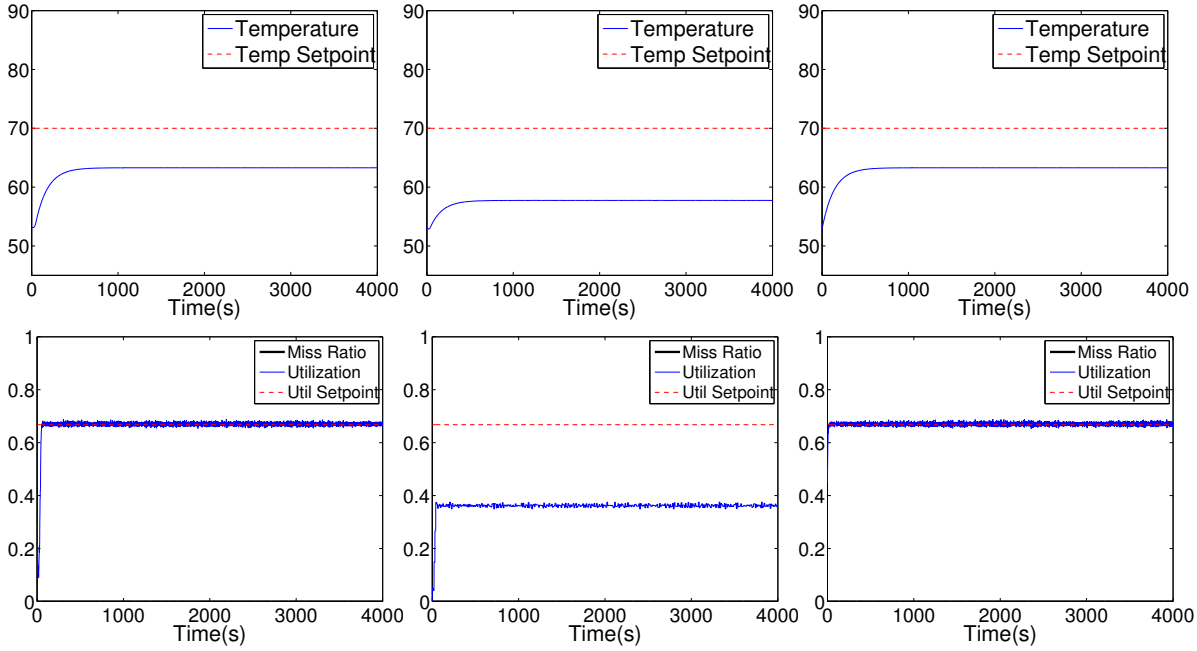


(a) TCUB (b) TC (c) FC-U

Figure 2.9: Performance comparison when etf is 2.

The results with $etf = 0.5$ are shown in Fig. 2.10. TCUB again successfully enforces the utilization bound, while the processor temperature remains below the set point. FC-U, like TCUB, enforces the utilization bound too. In contrast, TC significantly undershoots the utilization bound while its temperature remains significantly lower than the set point. This is caused by the fact that the task execution times are only half of the estimate. Note

such CPU underutilization results in unnecessarily low task rates, which are undesirable to applications.



(a) TCUB (b) TC (c) FC-U

Figure 2.10: Performance comparison when etf is 0.5.

Experimental results in these two sections demonstrate that TCUB is the only algorithm that can consistently maintain temperature set point and soft real-time performance under uncertainties of power consumption or the task execution times.

2.6.3 Robustness of TCUB

This experiment is designed to stress-test the robustness of TCUB under uncertainties of both the execution times and power consumption. For all experiments we plot the average temperature and utilization over the last 300 sampling period to exclude the transient response in the beginning of the experiment.

Fig. 2.11 demonstrates the robustness of TCUB when both the execution time factor and the power ratio vary in a wide region. The area full with circles labeled *empirical* represent the

simulations in which TCUB maintains satisfactory average temperature ($\leq 70.7°C, 1.01T_R$) and average utilization ($\leq 67.7\%, 1.01U_{max}$). The *theoretical* bound for the execution time factor is the maximum execution time factor below which the utilization controller maintains stability based on the analysis presented in [96]. The *theoretical* bound for the power ratio is the maximum power ratio under which the thermal controller can maintain stability based on Theorem 2. The *feasible* bound is determined based on the minimum task rates of our workload as discussed in Section 2.5.3. The area below both the theoretical bound and the feasible bound is the *stable* area of TCUB by theoretical analysis presented in Section 2.5.3. As shown in Fig. 2.11, the *empirical* area almost covers the analytical area. On the one hand, the results demonstrate that TCUB can maintain desirable temperature and utilization under considerable uncertainties in terms of both power consumption and execution times. On the other hand, close match between the analytical stable region and the empirical area demonstrate the effectiveness of our control model and analysis.

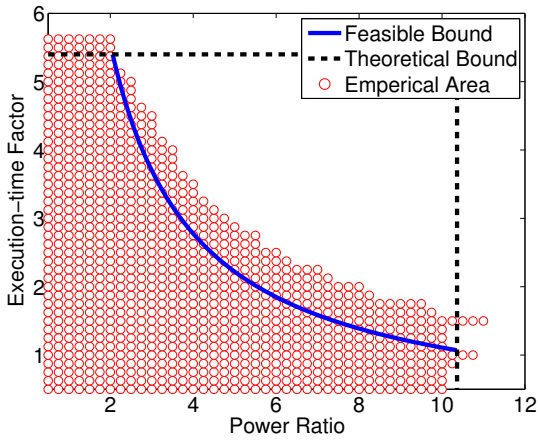
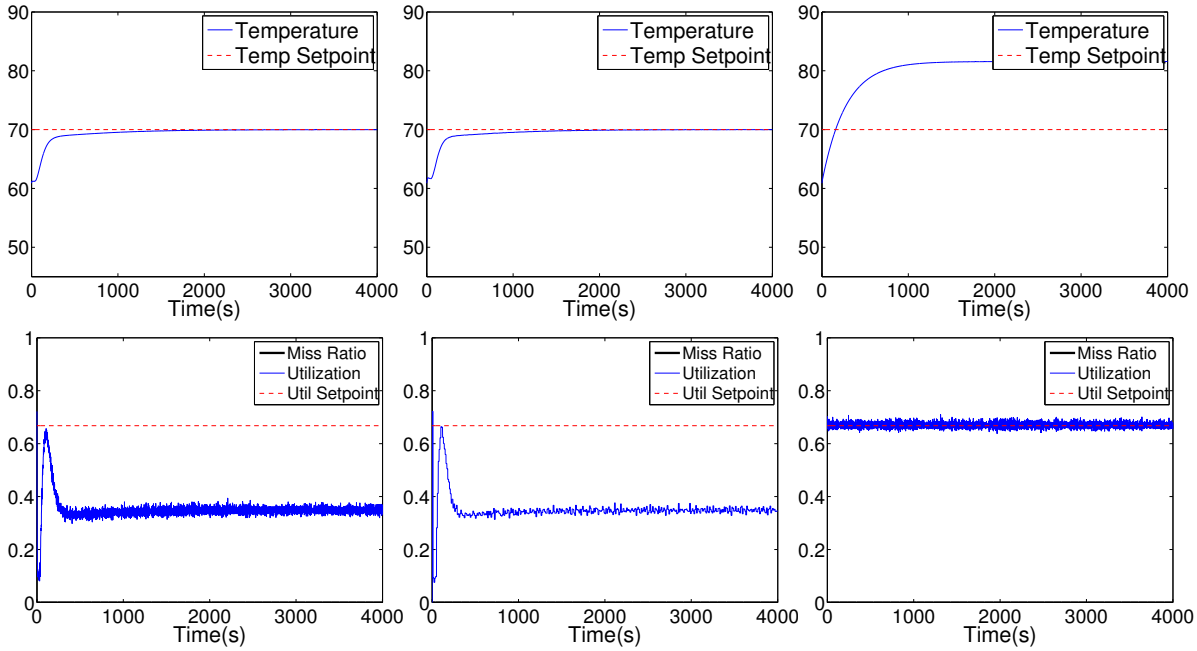


Figure 2.11: TCUB performance with varying power ratio and etf

2.6.4 Thermal Fault

This experiment is designed to examine the capability of TCUB to deal with thermal faults based on the empirical model presented in [23], we simulate the case fan failure by doubling the thermal resistance, R_{th}, of the processor. As shown in Fig. 2.12, under TCUB the temperature converges to $70°C$ while the utilization is considerably lower than the utilization set point. Since the thermal resistance doubles in this case, the processor generates more heat

at the same utilization. Therefore TCUB enforces the temperature set point by regulating the CPU utilization at a low level. TC performs similarly to TCUB as the utilization bound is not activated when the temperature converges to the set point. In contrast, FC-U significantly overshoots the temperature set point.

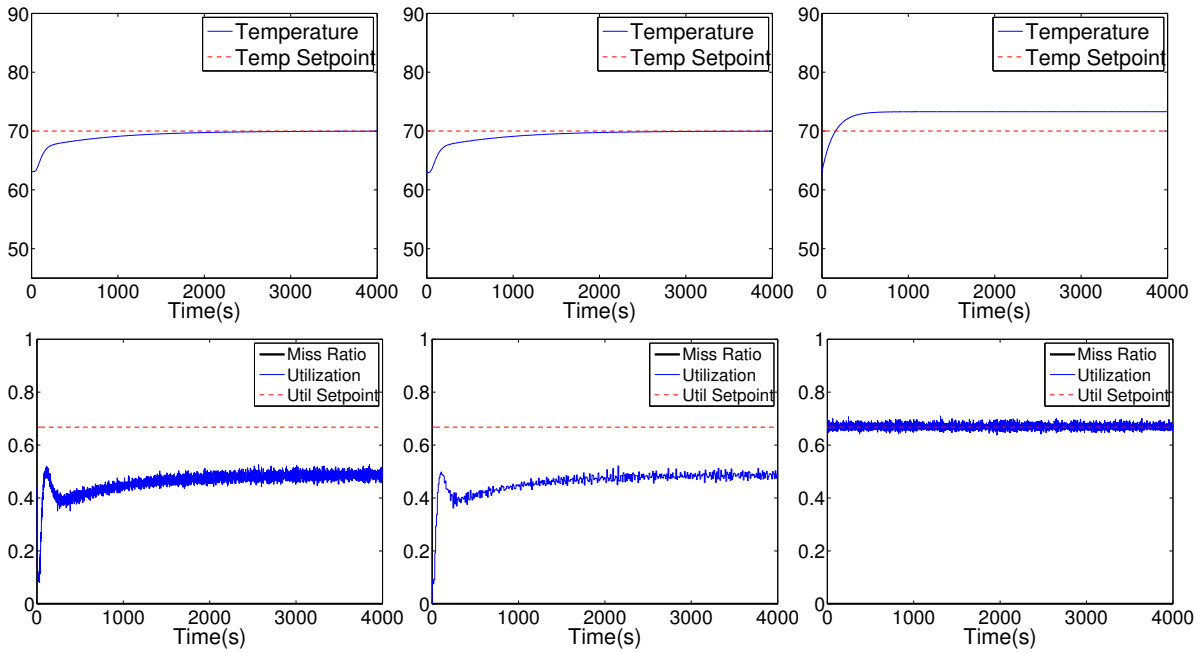


(a) TCUB (b) TC (c) FC-U

Figure 2.12: Performance comparison with thermal fault

2.6.5 Ambient Temperature Variation

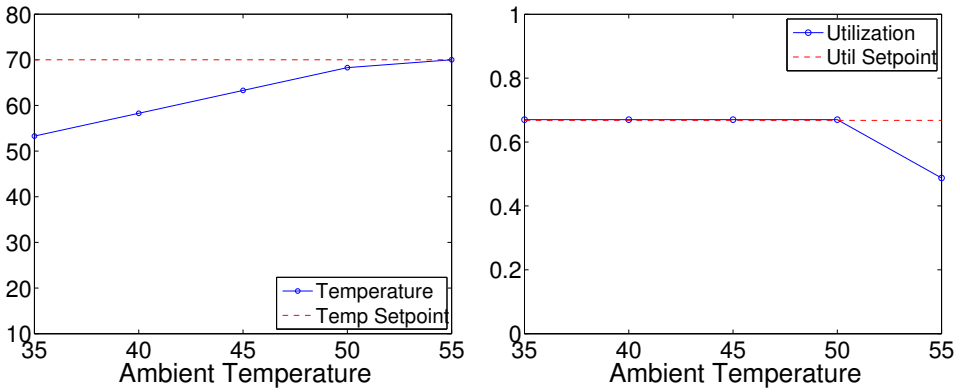
This experiment is designed to evaluate TCUB when the ambient temperature is higher than the default setting. Specifically, in the simulation we set the ambient temperature to $55°C$ which is higher than default setting by $10°C$. The power ratio and etf is fixed at 1.0 As shown in Fig. 2.13(a), TCUB tracks the temperature set point, while the utilization remains below the utilization bound. To offset the increase in the ambient temperature, TCUB lowers the CPU utilization so as to reduce the amount of heat generated by the processor. TC behaves similarly to TCUB. In contrast, FC-U exceeds the temperature set point at higher utilization.



(a) TCUB (b) TC (c) FC-U

Figure 2.13: Performance comparison with different ambient temperature

As claimed in Lemma 1, our thermal controller is robust to the uncertainty in the ambient temperature. The following experiment is designed to validate this claim by running a set of simulations under different ambient temperature in the range $[35, 55]$. In this experiment the power ratio and etf are preset to 1 and unchanged. Fig. 2.14 shows the experiments results. The temperature shown in Fig. 2.14(a) is the average of the processor's temperature in last 300 seconds of each simulation. When ambient temperature is lower than $50°C$, the steady state temperature of the processor stays below the temperature set point which utilization maintains the utilization set point. This is because the processor can not generate enough heat to surpass the temperature set point. In contrast when the ambient temperature is greater than $50°C$, the steady state temperature of the processor stays at $70°C$ while the utilization is reduced below the utilization set point in order to compensate for the higher ambient temperature. In both cases, the ambient temperature does not affect the steady temperature under TCUB, demonstrating the robustness of the thermal controller with regarding to ambient temperature.



(a) Temperature (b) Utilization

Figure 2.14: TCUB performance with varied ambient temperature

2.7 Summary

Many embedded systems face the critical challenge of managing both the processor temperature and software real-time performance in unpredictable environments. This paper presents TCUB, a control-theoretic algorithm for managing both the processor temperature and soft real-time performance. Rigorously modeled and designed based on control theory, TCUB can avoid processor overheating and maintain soft real-time performance. A salient feature of TCUB lies in its capability to handle different types of uncertainties in terms of (1) processor power consumption, (2) task execution times, (3) ambient temperature, and (4) unexpected thermal faults. The robustness of TCUB makes it particularly suitable for real-time embedded systems that must deal with highly unpredictable environments. Moreover, TCUB features a nested feedback control structure consisting of (1) a low-rate thermal controller dealing with the slower thermal dynamics, and (2) a high-rate utilization controller handling the faster CPU utilization dynamics caused by uncertainties in task execution times. The nested control scheme is modular, efficient, and practical for embedded systems with tight resource constraints. The advantages of TCUB have been demonstrated through extensive simulations under a broad range of system and environmental conditions.

Chapter 3

Feedback Thermal Control for Real-time Systems on Mulitcore Processors

3.1 Introduction

Embedded real-time systems face significant challenges in thermal management as they adopt modern computing platforms with increasing power density. While traditional embedded real-time systems typically run on single-core low-power microcontrollers, the increasing complexity of real-time applications demands the adoption of modern multicore microprocessors to leverage their computing power. Such systems must avoid processor overheating while maintaining desired real-time performance. The need to enforce temperature bounds can conflict with the need to meet real-time performance requirements, because thermal management mechanisms such as Dynamic Voltage and Frequency Scaling (DVFS) reduce processor speed resulting in prolonged execution times for real-time tasks. While modern processors usually rely on hardware throttling mechanisms to prevent overheating, such mechanisms can cause severe performance degradation unacceptable to real-time applications. Moreover, modern processors can exhibit significant *uncertainties* in their power and thermal characteristics. For instance, the power consumption of a processor may vary significantly when running different applications due to the different sets of instructions executed [46].

In recent years, control-theoretic thermal management approaches have shown promise in [20, 27, 28, 54, 99, 105, 106] handling uncertainties in thermal characteristics. In contrast to

heuristic-based design relying on trial-and-error, control-theoretic approaches provide a scientific framework for systematic design and analysis of thermal control algorithms. However, previous research on feedback thermal control for embedded real-time systems focused on single-core processors and cannot handle the practical limitations of multicore processors. Thermal management mechanisms such as DVFS only support a finite set of states, leading to *discrete* control variables that cannot be handled by standard linear control techniques. Moreover, multicore processors require the temperatures and real-time performance of *multiple* cores to be controlled simultaneously, leading to multi-input-multi-output (MIMO) control problems with inter-core thermal coupling.

We present *Real-Time Multicore Thermal Control (RT-MTC)*, a novel feedback thermal control algorithm specifically designed to meet the challenges posed by multicore processors. RT-MTC employs a feedback control loop that enforces the desired temperature and CPU utilization bounds of embedded real-time systems through DVFS. RT-MTC employs an efficient and robust control design that integrates three components.

- a robust nonlinear proportional controller that deals with uncertainties in power consumption;
- a saturation block for the controller output that enforces the schedulable utilization bound;
- a Pulse Width Modulation (PWM) component that achieves desired control input by dynamically switching between discrete voltage/frequency levels.

RT-MTC combines a control-theoretic approach and a practical design. In contrast to heuristics-based solutions relying on extensive testing and hand tuning, we provide control-theoretic analysis of the stability and robustness of RT-MTC under uncertainties in power consumption. At the same time, RT-MTC employs a simple and efficient control algorithm suitable for run-time execution. Moreover, RT-MTC can be easily implemented in the user space without modification to the OS kernel which is usually required by traditional thermal-aware real-time scheduling approaches. The robustness and advantages of RT-MTC over existing thermal control approaches are demonstrated through implementation on Linux and experiments on an Intel Core 2 Dual processor as well as extensive simulations with varying power consumption.

The rest of the chapter is organized as follows. Section 3.2 formulates the problem of thermal contorl for real-time systems on a multicore processor. Section 3.3 outlines the structure of RT-MTC. Section 3.4 presents a model that characterizes the thermal dynamics of real-time systems. Section 3.5 details the design and stability analysis of RT-MTC. Section 3.6 describes some detail to implement RT-MTC. Section 3.7 provides simulation results. Section 3.8 introduces related work. Section 3.9 provide summary of the this chapter.

3.2 Problem Formulation

We assume a common embedded real-time system model where the workload consists of real-time tasks released periodically. A embedded real-time system comprises a set of periodic real-time tasks running on a multicore processor with m homogeneous cores. The processor supports Dynamic Voltage and Frequency Scaling (DVFS). We assume two common characteristics of DVFS in mainstream multicore processors (e.g., Intel Core2, i5, i7 and Atom). First, the frequency and voltage of all the cores can only be scaled *uniformly*, i.e., all cores always share the *same* frequency and voltage. Second, the processor only supports a *discrete* set of frequencies. New challenges are posed by The dicretization and nonlinearity introduced by both assumptions pose key challenges to thermal control design that were not addressed in previous works [71, 99, 104–106].

We assume *partitioned* multicore real-time scheduling, under which tasks are statically partitioned and bound to processor cores. There is a real-time tasks set $\mathbb{S}$ with n independent, periodic real-time tasks for the processor. For core l, there is a task set $\mathbb{S}_l \subseteq \mathbb{S}$ with n_l real-time tasks. Each task s_i in the task set $\mathbb{S}_l$ has a period p_i, a soft deadline d_i, and a worst-case execution time c_i. The utilization of an individual core l is thus $U_l = \sum_{s_j \in \mathbb{S}_l} \frac{c_j}{p_j}$.

We assume the tasks on a core are scheduled locally based on a real-time scheduling policy with a known schedulable utilization bound U_b, e.g., Rate Monotonic (RM) or Earliest Deadline First (EDF) under certain conditions [56]. The tasks on a core l meet their deadlines

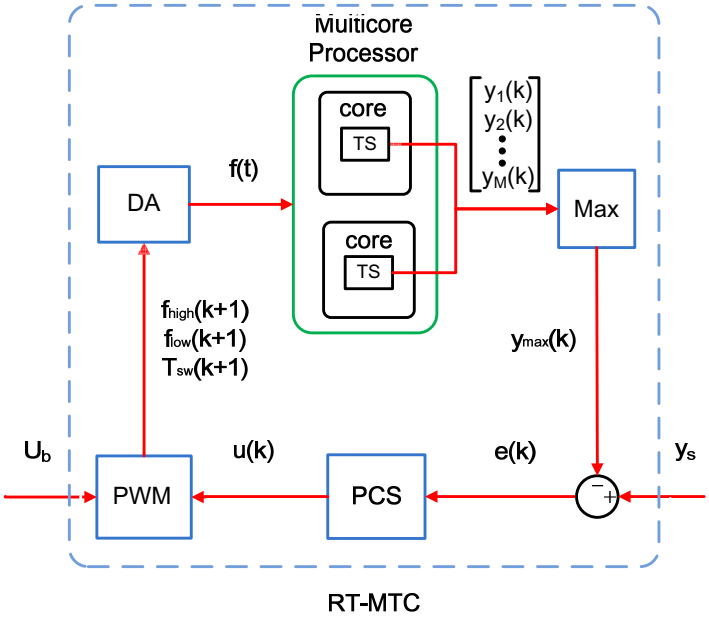


Figure 3.1: Feedback control loop of RT-MTC

if $U_l \leq U_b$. The system can therefore guarantee the schedulability of all the tasks on a core by enforcing the schedulable utilization bound. [4]

Given a embedded real-time system running on a multicore processor, our problem is to control the temperature of the processor such that the *maximum* temperature among all the cores tracks a temperature set point, y_s, subject to the constraint of utilization bound U_b on each processor core. The temperature set point y_s is the desired temperature below the maximum temperature tolerable by the processor. Our control problem formulation therefore aims to meet both the thermal and real-time performance requirements of a embedded real-time system.

3.3 Overview of RT-MTC

The feedback control loop of RT-MTC, shown in Fig. 3.1, consists of a Temperature Sensor (TS) for each core, a Proportional Controller with Saturation (PCS), Pulse Width Modulation (PWM), and a DVFS Actuator (DA). The user input to RT-MTC is the desired temperature set point y_s and the utilization bound U_b. The feedback control loop is invoked periodically at the end of every sampling period. Specifically, at the end of k^{th} sampling period, RT-MTC performs the following operations:

1. The TS on each core measures the temperature of the core i, $y_i(k)$. The Max function calculates the maximum temperature among all cores and feeds the maximum temperature $y_{\max}(k)$ among all the cores to the PCS.

2. The PCS computes the controller output $u(k)$ as follows:

$$u(k) = \begin{cases} 1, \text{ if } k_p e(k) > 1, \\ -1, \text{ if } k_p e(k) < -1, \\ k_p e(k), \text{ otherwise;} \end{cases} \tag{3.1}$$

   where k_p is the coefficient of proportional control and $e(k) = y_s - y_{\max}(k)$. The output of the controller is limited to the range $[-1, 1]$. The PCS design is discussed in more details in Section 2.5.3.

3. The PWM receives the controller output $u(k)$ and calculates a pair of frequencies $f_{high}(k+1), f_{low}(k+1)$ and the switching time $T_{sw}(k+1)$. Details of calculating $f_{high}(k+1), f_{low}(k+1), T_{sw}(k+1)$ are presented in Section 3.4.2.

4. The DA adjusts the frequency of the multicore processor via the DVFS interface according to the $(f_{high}(k+1), f_{low}(k+1), T_{sw}(k+1))$ input from the PWM. Specifically, at $T_{sw}(k+1)$ seconds after the beginning of the current sampling period, the processor switches its frequency from $f_{high}(k+1)$ to $f_{low}(k+1)$. The implementation of DA is detailed in Section 3.6.

[4]Our approach can be extended to support a mixed task set containing periodic and soft real-time aperiodic tasks via well known aperiodic server mechanisms [57] by enforcing appropriate schedulable utilization bounds.

3.4 Thermal Dynamic Model

As the first step of control design and analysis, we now present a difference equation model to characterize the relationship between the frequency and the temperature. We construct the model in three steps. We first caputre the power consumption. Based on a well known power model, we then characterize the impact of PWM on the power consumption model. Finally, we complete the system model by incorporating a widely used thermal RC model that characterizes the relationship between power consumption and temperature.

We note that our system model is necessarily a simplification of the actual system's thermal behavior for the purpose of control-theoretic design and analysis. The inherent robustness of feedback control enables our system to handle considerable modeling errors in model parameters, as demonstrated in our evaluation (Sec. 3.7.1).

3.4.1 Power Model

As shown in [28], the average power $\bar{P}(k)$ of a core in the k^{th} sampling period can be modeled as

$$\bar{P}(k) = U(k)P_{\text{act}}(k) + (1 - U(k))P_{\text{idle}}(k)$$

where $U(k)$ is the CPU utilization of the core, $P_{\text{act}}(k)$ is the active power, and $P_{\text{idle}}(k)$ is the idle power in k^{th} sampling period. $P_{\text{idle}}(k)$ can be approximated by a piecewise linear model $P_{\text{idle}} = (C_0(V(k)) + C_1(V(k))y(k))V(k)$ [77]. A well-known model of the active power is $P_{\text{act}}(k) = C_2V^3(k)$, where C_2 is a constant coefficient and $V(k)$ is the supply voltage [79].

We can rewrite the average power as

$$\bar{P}(k) = \bar{P}_a(k) + C_y y(k) \tag{3.2}$$

where $\bar{P}_a(k) = U(k)C_2V^3(k) + C_0(V(k))V(k)$ and $C_y = C_1(V(k))$. $\bar{P}_a(k)$ and C_y can be expressed in terms of the frequency, based on the relationship between supply voltage and frequency, $V(k) = Kf(k) + V_{\text{th}}$ [52] and $\frac{U(k)}{f(k)} = \frac{U_0}{f_0}$ where U_0 and f_0 are the initial CPU utilization and frequency. Note we assume that the processor utilization scales proportionally with the frequency which usually hold for those CPU bound applications.

3.4.2 Pulse Width Modulation (PWM)

As each core of the multicore processor runs under a discrete set of frequencies, the power $\bar{P}_a(k)$ in equation (3.2) can only switch between discrete levels. To track the temperature set point closely, PWM is employed to map desired average power in each sampling period to the discrete frequency levels supported by the processor.

The continuous input to the PWM in the k^{th} sampling period is $u(k) \in [-1, 1]$. The PWM computes $(f_{high}(k+1), f_{low}(k+1), T_{sw}(k+1))$ based on $u(k)$. The upper limit of the output corresponds to the maximum frequency supported by the processor. The lower limit of the output corresponds to the lowest frequency that satisfies the utilization bound or the minimum frequency, whichever is higher. Let the frequency corresponding to the upper and lower limit of $u(k)$ be $f_{\max}, f_{\min}$, and let $f_u(k) = f_{\min} + (f_{\max} - f_{\min})\frac{u(k)+1}{2}$. To minimize the change in CPU speed, PWM first chooses a pair of *consecutive* frequency levels f_i and f_{i+1} which satisfy $f_i \leq f_u(k) \leq f_{i+1}$ from the supported discrete frequency set; these are designated $f_{low}(k+1)$ and $f_{high}(k+1)$ respectively. The time to switch from $f_{high}(k+1)$ to $f_{low}(k+1)$ is computed as

$$T_{sw} = \frac{f_u(k) - f_{low}(k+1)}{f_{high}(k+1) - f_{low}(k+1)} T_s,$$

where T_s is the sampling period. Note if $f_u(k)$ equals any frequency in the supported frequency set, both $f_{high}(k+1), f_{low}(k+1)$ will exactly equals that frequency and $T_{sw} = 0$.

Let $\bar{P}_{a,max}$, $\bar{P}_{a,min}$ be the upper and lower bound of $\bar{P}_a$, which are the average power consumption at $f_{\max}$ and $f_{\min}$, respectively. We can rewrite the power model to incorporate PWM based on (3.2) as

$$\bar{P}(k) = G_p(P_{ap}u(k) + P_{am}) + C_y y(k) \tag{3.3}$$

where $P_{ap} = (\bar{P}_{a,max} - \bar{P}_{a,min})/2$, $P_{am} = (\bar{P}_{a,max} + \bar{P}_{a,min})/2$, and G_p is the gain to represent the uncertainty caused by power variation.

The power consumption model (3.3) approximates the power behavior of the processor, since it derives the average power rather than actual power. However, as we shown in our stability analysis (Section 2.5.3) and experiments (Section 3.7.1), the inherent robustness of

our feedback control design can tolerate considerable modeling error without compromising system stability.

3.4.3 Thermal Dynamic Model

Our control design is based on a well-established thermal RC model for multicore processors with M cores and a heat sink [24]. Compared to architecture-level thermal models such as Hotspot [41], the model presented here is simpler but more suitable for control design of thermal management. The effectiveness of the model has been validated in [24, 79].

Symbol	Meaning
$R_i, R_h, R_a, R_{i,j}$	thermal resistance of the core i, the heat sink, environment and thermal resistance between the core i and j
C_i, C_h	thermal capacitance of the core i and the heat sink
y_0, y_i, y_h	temperature of environment, the core i and the heat sink
P_i	power of the core i
$\mathbb{N}_i$	the set of cores adjacent the core i

Table 3.1: Symbols in thermal dynamic model

Based on the symbols listed in Tab. 3.1, the thermal dynamic model of the multicore processor can be written in the following compact form:

$$\dot{\mathbf{Y}}(t) = A\mathbf{Y}(t) + B_P\mathbf{P}(t) + B_y y_0 \tag{3.4}$$

where $\mathbf{Y}(t) = [y_1(t), \ldots, y_M(t), y_h(t)]^\mathsf{T} \in \mathbb{R}^{M+1}$, $\mathbf{P}(t) = [P_1(t), \ldots, P_M(t)]^\mathsf{T} \in \mathbb{R}^M$ and y_0 is the ambient temperature, $A \in \mathbb{R}^{(M+1)\times(M+1)}$, $B_P \in \mathbb{R}^{(M+1)\times M}$ and $B_y \in \mathbb{R}^{(M+1)}$. The

matrices A, B_P and B_y are computed as follows:

$$A(i,j) = \begin{cases} \frac{-1}{C_i}\left(\frac{1}{R_i} + \sum_{m\in\mathbb{N}_i}\frac{1}{R_{i,m}}\right), \text{ if } i = j \neq (M+1) \\ \frac{1}{R_{i,j}C_i}, \text{ if } j \in \mathbb{N}_i \\ \frac{1}{R_iC_i}, \text{ if } i \neq (M+1) \text{ and } j = (M+1) \\ \frac{1}{R_jC_h}, \text{ if } i = (M+1) \text{ and } j \neq i \\ \frac{-1}{C_h}\left(\frac{1}{R_a+R_h} + \sum_{m=1}^{M}\frac{1}{R_m}\right) \text{ if } i = j = (M+1) \\ 0, \text{ otherwise.} \end{cases},$$

$$B_P(i,j) = \begin{cases} \frac{1}{C_i}, \text{ if } i = j \\ 0, \text{ otherwise.} \end{cases},$$

$$B_y(i) = \begin{cases} \frac{1}{C_h(R_a+R_h)}, \text{ if } i = M+1 \\ 0, \text{ otherwise.} \end{cases}.$$

We use a Zero Order Hold (ZOH) equivalent model [25] in which the average power-model for $\bar{P}(k)$ is assumed to be held constant and the average environmental temperature is $y_0(k) = \frac{1}{T_s}\int_{kT_s}^{(k+1)T_s} y_0(t)dt$ during the k^{th} sampling period. The ZOH equivalent of (3.4) is

$$\mathbf{Y}(k+1) = \Phi_o\mathbf{Y}(k) + \Psi_P\bar{\mathbf{P}}(k) + \Psi_y y_0(k) \tag{3.5}$$

where $\Phi_o = e^{AT_s}$, $\Psi_P = \left(\int_0^{T_s} e^{A\tau}d\tau\right)B_P$, $\Psi_y = \left(\int_0^{T_s} e^{A\tau}d\tau\right)B_y$ and $\bar{\mathbf{P}}(k) = [\bar{P}_1(k), \ldots, \bar{P}_M(k)]^\mathsf{T} \in \mathbb{R}^M$. Substituting the power model (3.3) for $\bar{P}(k)$ in (3.5) results in:

$$\mathbf{Y}(k+1) = \Phi\mathbf{Y}(k) + P_{ap}\Psi_P G_p u(k) + \Psi_y y_0(k) + P_{am}\Psi_P G_p \tag{3.6}$$

in which $\Phi = \left(\Phi_o + C_y\Psi_P\begin{bmatrix} I_M & 0 \end{bmatrix}\right)$ where $\begin{bmatrix} I_M & 0 \end{bmatrix}$ $\in \mathbb{R}^{M\times(M+1)}$ and $I_M \in \mathbb{R}^{M\times M}$ denotes the identity matrix. The term involving $y_0(k)$ relates how environmental temperature changes can perturb the system. The last term represents a fixed-disturbance due to the mean active power resulting from our proposed modulation approach.

In practice the model parameters can be estimated using well-known system identification method. Essentially, there are two methods to acquire the parameters of the compact thermal model. We can either extract the parameters based on fine grain thermal RC models, for example Hotspot [41] or estimate the parameters using realistic operational data, which is also the method we used in this paper. The detailed description of model identification is presented in Section 3.7.1.

3.5 Control Design

We propose a low-complexity controller to tackle the problem of thermal management of real-time systems on multicore processors. Our control design ensures that the maximum temperature of the cores tracks the thermal set-point without violating the utilization constraints. Although the control structure shown in Fig. 3.1 only has single input, the PCS must control the temperature of multiple cores simultaneously. Previous approaches to thermal control for the single core processor [28] is not suitable to multicore thermal control because their control design do not handle the interaction among the thermal dynamics of different cores. In this section we present a control design which can handle not only thermal coupling among cores but also other nonlinearities induced by the multicore processors.

3.5.1 Stability Analysis and Control Design

The PCS is designed based on passivity [76] and can accommodate the nonlinearities induced by the *Max* function and the saturation. There are various precise mathematical definitions for passive systems that essentially state that the output energy must be bounded so that the system does not produce more energy than was initially stored. Under certain technical conditions, strictly input and strictly output passive systems are Lyapunov stable [87]. In this case, passivity offers advantages for computing a Lyapunov function that is used to prove stability of the closed-loop system.

In order to analyze the stability of RT-MTC, we assume that the set-point $T_b = 0$ and we consider the unperturbed system where $y_0 = 0$, $\Psi_P G_P = 0$ in (3.6). We provide sufficient

conditions that ensure the existence of a Lyapunov function for the closed loop system, and thus, stability of the RT-MTC. A detailed proof can be found in [29]. The disturbance in the power model arises because of (1) the ambient temperature that can change but is measurable and (2) the mean active power introduced by the PWM. We can minimize the steady-state error by taking into account these terms in the set-point T_b (the detailed derivation of T_b can be found in [29]).

Theorem 3. *Consider the closed-loop system shown in Fig. 3.1 with $T_b = 0$ and assume that the power model of the multicore processor is described by (3.6) with $y_0(k) = 0$ and $P_{am}\Psi_P G_P = 0$. If there exists a matrix $P = P^\mathsf{T} > 0$ and $-\infty < \delta < 0$ such that the following LMI is satisfied:*

$$\begin{bmatrix} \Phi^\mathsf{T} P \Phi - P & \Phi^\mathsf{T} P P_{ap} \Psi_P G_p - \frac{1}{2} C_l^\mathsf{T} \\ \left(\Phi^\mathsf{T} P P_{ap} \Psi_P G_p - \frac{1}{2} C_l^\mathsf{T}\right)^\mathsf{T} & \delta + P_{ap}^2 G_p^\mathsf{T} \Psi_P^\mathsf{T} P \Psi_P G_p \end{bmatrix} \leq 0 \tag{3.7}$$

for all $l \in \{1, \ldots, M\}$, where C_l is the coefficient for the measured temperature of the core l, then the closed-loop system is passive and stable.

By exploring the solution of the LMI (3.7) given in Theorem 3, we can acquire the stability condition of the system under modeling error. Specifically, for items in the search space of power gain, thermal resistance and capacitance, we can check whether the LMI is solvable and then decide whether the closed-loop system is stable with the parameters. Accordingly we derive robustness of the system in terms of the range of uncertain parameters, power gain and thermal related parameters resulting in stable systems.

The above theorem can also be used for designing the controller. This is achieved by finding the smallest value of δ that satisfies the LMI (3.7), The controller gain of the PCS (equation (3.6)) is defined as $k = -\frac{1}{\delta}$. This is the highest proportional gain that guarantees stability of the closed-loop system. In general, higher controller gain improves control performance. If there is deviation from the set point, high gain controller ensures that the system will converge to the set-point as fast as possible. The LMI shown in the theorem can be solved efficiently using standard LMI tools such as the Matlab LMI toolbox and the Scilab lmitool.

3.6 Implementation of RT-MTC

We have implemented RT-MTC on top of Linux, using a combination of Python, MATLAB, and C. The PCS, PWM, DVFS Actuator, and Max components shown in Fig. 3.1 are written in Python.

All the components in the feedback control loop are implemented in one process assigned the highest real-time process priority so that RT-MTC can be executed periodically with minimum interference from real-time tasks.

Thermal Sensor: Most modern multicore processors are equipped with hardware thermal sensors for each individual core, which are supported by the operating system or third-party libraries. For example, the temperature of cores can be read from the interface provided by *lmsensor* [3] via the coretemp driver (/sys/bus/platform/drivers/coretemp/) in Linux. The thermal information can also be acquired from standard ACPI interfaces. For those multicore processors without thermal sensors on each core, such as those used in embedded systems, soft thermal sensors [51] can be employed to estimate the temperature of a single core.

PCS and PWM: The implementations of PCS and PWM are straightforward, based on the description in Sec. 3.5 and Sec. 3.4.2.

DVFS Actuator: We implemented the DVFS Actuator using the *signal* mechanism provided by POSIX interface. First, an alarm is set to be fired at the switching time T_{sw} by using the POSIX *alarm* function. When the alarm expires, a $SIGALRM$ signal is sent to the process's signal handler set by the function *sigact*. The signal handler calls a procedure to switch the frequency of the multicore processor from the high level f_{high} to the low level f_{low} via a interface which can access the processor's DVFS function, for examples, ACPI, lmsensor or Machine Specific Register. The delay between PWM output switching time T_{sw} and the time that the frequency is actually switched relies on the resolution of clock interrupt of the underlying operating system. For example, the Linux kernel uses a configurable time resolution (known as *jiffy*) which ranges from $1ms$ to $10ms$. Even at a resolution of $10ms$, the delay has negligible effect on the control performance, since it is comparatively much shorter than the sampling period. We choose $10s$ as the sampling period in our implementation

because it is short enough to control the thermal behavior of the processor , which has time constant greater than $100s$, without imposing singnificant overhead from frequency switching and computation.

3.7 Evaluation

We first evaluate RT-MTC through experiments based on above implementation and then perform extensive simulations with parameters acquired from model identification experiments. An Intel Core 2 Duo two core processor is used to run the experiments and be the target of simulations as it provides discrete DVFS mechanism. Moreover thermal parameters, especially thermal capacitance, of Intel Core2 Duo are acquired directly as shown later. The simulations complement experimental results by allowing us to examine RT-MTC's performance under stress-test conditions (such as fan failure) which are difficult or dangerous to run on real hardware.

3.7.1 Experiments

The hardware platform used for the experiments is a Lenovo W500 laptop with an Intel T9400 Core 2 Duo dual core processor and the Linux kernel 2.6.32 distributed with Fedora 12.[5] The T9400 processor has 2 digital thermal sensors located on each core and supports processor-wide DVFS, that is, the two cores' frequencies must be set uniformly. The DVFS frequencies and the thermal properties of the T9400 are listed in Table 3.2.

Model Identification

To acquire the parameters of the thermal RC model, we first run a set of real-time workloads to profile the processor's thermal behavior. Then the thermal parameters is identified from the experiments results by Matlab Model Identification Toolbox. The real-time workloads

[5]Although we only present the results of experiments for a dual core processor, the methodology and implementation can be extended to the processor with more than two cores easily since control design proposed in this paper is based on a general multicore processor model.

Properties	Value
Frequency	2.53, 1.6, 0.8 GHz
Voltage	1.175, 1.00, 0.900 V
T_{junc}	$105°C$
Thermal Design Power (TDP)	35W

Table 3.2: Frequencies and thermal properties of the T9400 processor.

used for model identification involves two micro benchmarks, CRC and Bzip2. CRC is a data verification application chosen from Mibench [36], a test suite for embedded systems. Bzip2 is a data compression tool chosen from SPEC CPU 2006 [2], a standard benchmarks suite. We implement three kinds of workloads: CRC alone, Bzip2 alone and a Mixed workload containing both microbenchmarks. The workload for each core is identical and involves 5 periodic tasks which are either CRC or Bzip2 according to the type of the workload. The deadlines of the tasks are set to the same as their periods. The periods and execution time of the tasks are listed in Table 3.3.

	Task 1	Task 2	Task 3	Task 4	Task 5
Period	250	300	450	500	1000
Execution Time	23	27	41	45	90

Table 3.3: Workload tasks period and execution time when frequency is 2.53GHz (ms).

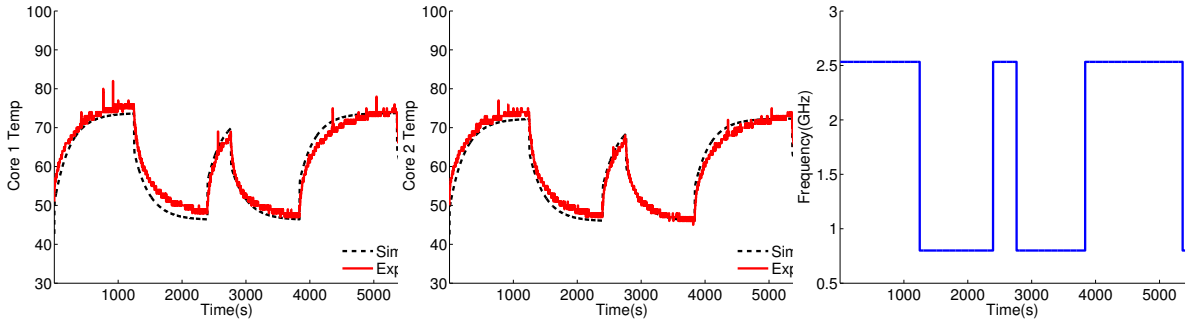
Thermal parameters (Mixed, Fit*: 82%)						
$R_1(\Omega)$	$C_h(F)$	$R_{12}(\Omega)$	$R_2(\Omega)$	$C_2(F)$	$C_1(F)$	$R_a + R_h(\Omega)$
1.61	216.74	16.16	1.46	1.25	1.25	1.05
Thermal parameters (Bzip2, Fit:83%)						
$R_1(\Omega)$	$C_h(F)$	$R_{12}(\Omega)$	$R_2(\Omega)$	$C_2(F)$	$C_1(F)$	$R_a + R_h(\Omega)$
1.35	263.02	15.23	1.13	1.61	1.61	1.35
Thermal parameters (CRC, Fit: 81%)						
$R_1(\Omega)$	$C_h(F)$	$R_{12}(\Omega)$	$R_2(\Omega)$	$C_2(F)$	$C_1(F)$	$R_a + R_h(\Omega)$
1.78	242.23	16.83	1.56	1.35	1.35	1.08

*: the accuracy index in Matlab model identification Toolbox.

Table 3.4: Results of model identification

To capture the comprehensive thermal behavior for different frequencies, we employ a pseudo-sequence of frequency as input, where frequency switches between $2.53GHz$ and $0.8GHz$. Considering the large time constant of the processor's thermal behavior, we run each workload for 5400s. Table 3.4 shows the results of the model identification via Matlab Model Identification Toolbox. Fig. 3.2 illustrates the temperature and frequency of the Mixed workload; the other two workloads are omitted here due to space constraints.

There are two important observations from Table 3.4. First, it indicates the efficacy of the thermal dynamic model, as the estimated model parameters result in fitness levels above 80% for all three workloads. Second, the model parameters estimated under different workload differ considerably. This entails that thermal control must be robust against uncertainties of model parameters caused by different workloads since it is unrealistic to expect users to re-estimate the parameters via system identification for every workload. Such robustness against modeling errors is an important advantage of RT-MTC, as shown in both the empirical results and the simulation study presented below.



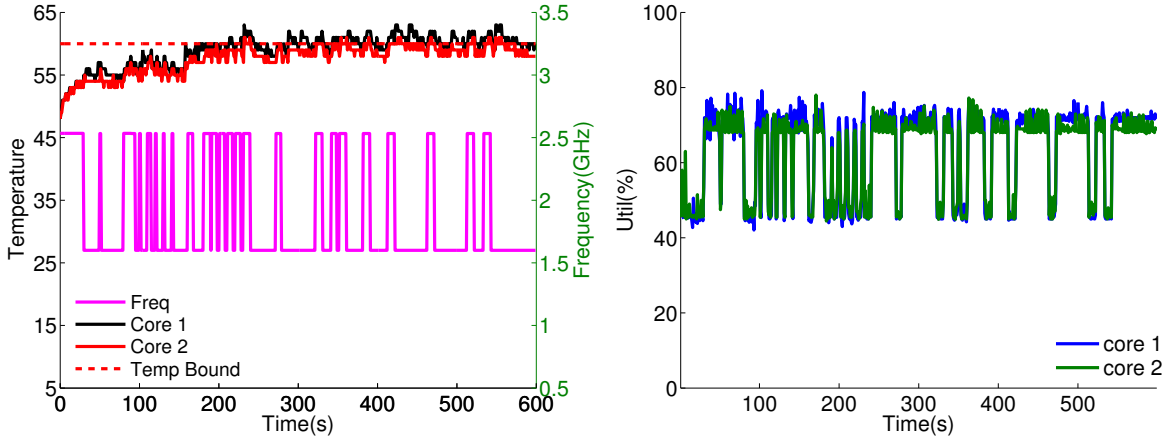
(a) Core 1 Temperature (b) Core 2 Temperature (c) Frequency

Figure 3.2: Model identification data (mixed workload)

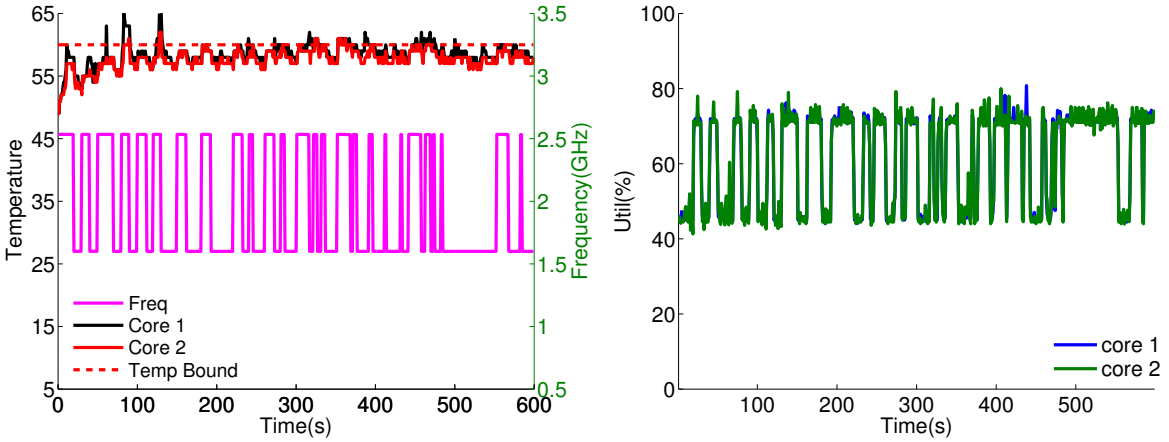
Experiment Results

In this section we present the experimental results of RT-MTC on the real hardware platform. We run RT-MTC under the workload of the CRC and the Mixed for 10 minutes each. The controller parameters of RT-MTC are computed using the thermal RC model parameters of the Mixed workload. In this experiment we choose the temperature set point as $60°C$ to

ensure that internal thermal throttling circuit is not activated even when there is overshoot during temperature adjustment.



(a) Mixed Workload



(b) CRC

Figure 3.3: Experimental results of RT-MTC

Two important observations can be made from the results plotted in Fig. 3.3. First, RT-MTC enforces both the temperature set point and the utilization bound. As seen in Fig. 3.3(b), after $280s$ the temperature is steady at the temperature set point, $60°C$. The average upper limit of the utilization is 74%, which is below the utilization bound. Second, RT-MTC (with the same control parameters) can control the thermal behavior of the processor effectively under *both* test workloads. As shown in Table 3.4, there is difference between the parameters identified by the Mixed and the CRC workloads, which induces modeling

error. Ensuring temperature set point in both cases shows RT-MTC robustness against modeling error induced by different workloads. Although there are spikes in temperature during the CRC workload caused by background services (which cannot be manipulated by our user-space implementation), RT-MTC quickly counteracts these spikes.

3.7.2 Simulation

We perform extensive simulations based on the model parameters identified from the experiments in Sec. 3.7.1. Although we wish to explore the performance of RT-MTC in extreme scenarios, it is often impractical to carry such experiments out on real hardware. For example, an experiment int RT-MTC's performance in the face of fan failure would be likely to damage the processor. For this reason, we stress-test the performance of RT-MTC under simulation, as discussed in this section.

Simulation Setup

There are two components in our simulation environment: an event driven simulator implemented in C++ and a Simulink module implemented in MATLAB (R2008a). The C++ simulator simulates embedded real-time systems over multicore processors and calculates the processor utilization according to the frequency output by the controller. The Simulink module performs the controller's computation. And the Simulink module also calculates the temperatures of multicore processors based on the utilization generated by the C++ simulator. The C++ simulator and the Simulink module communicate with each other through a TCP connection.

The target multicore processor in our simulation is the dual core processor, Intel Core 2 Duo T7200 [1]. The power and thermal related parameters of T7200 are shown in Table 3.5. The parameters of the leakage power model are acquired by linear approximation of an accurate leakage power model [58]. The active power and available frequencies are obtained from Intel T7200 data sheet [1]. Note that although the evaluation is only preformed on the dual-core processor, our approach for thermal management is developed for general multicore processors and therefore can handle the processors with more cores.

We use the same methodology and tools for model identification as described in Sec. 3.7.1. The acquired thermal parameters are listed in Table 3.5. As thermal design is different between manufacturers, it is reasonable that these parameters identified vary significantly from those identified for the T9400.

Power Parameters				
f(GHz)	0.8,	1.2,	1.6,	2.0
C_0	-0.3638,	-0.3687,	0.1071,	2.3367
C_1	0.0191,	0.0342,	0.0608,	0.1066
C_2	7.7378			

Thermal Parameters					
$R_1(\Omega)$	0.53	$C_h(F)$	390	$R_{12}(\Omega)$	5.5
$R_2(\Omega)$	0.57	$C_2(F)$	39.14	$C_1(F)$	50.38
$R_a + R_h$	0.2				

Table 3.5: Simulation parameters

In the simulations we use a fine-grained workload which runs 10 periodic soft real-time tasks on each core. We assume partitioned scheduling for the multicore embedded real-time systems. The Rate Monotonic (RM) scheduling algorithm [56] is employed to schedule all tasks on each core. The utilization bound is set to 0.71. At the beginning of the experiment, the period of each task T_i is randomly generated in the range $[100ms, 200ms]$. The execution time of each task is generated to keep each task's utilization nearly equal and the sum of all tasks' utilization at 0.7, just below the utilization bound.

In the following simulations, we set the temperature bound to $60°C$, below the temperature achieved by the Thermal Design Power (TDP) of T7200 so as not to activate the internal hardware thermal regulation. Note that the effectiveness of our approach does not rely on the specific temperature bound.

We compare RT-MTC against four other baseline algorithms, OPEN, Reactive, MPC-QUAN and MPC-PWM. The algorithm OPEN statically sets the processors' frequency at beginning of the simulation and does not change it while the simulation runs.

MPC-QUAN and MPC-PWM are control-theoretic approaches and based on the algorithm proposed in [106]. The control algorithms of both baselines are the solutions of the following

constraint optimizing problem with the optimizing objective as follows:

$$J(k) = \sum_{i=1}^{H_p} |y_{max}(k+i) - y_s|^2 \tag{3.8}$$

where H_p is the prediction horizon and y_s is the temperature set point. The solution of the optimizing problem also needs to satisfy the constraints of the utilization bound, the thermal bound, and the frequency limit. Note that $T(k)$ must follow the thermal model (3.5). The solution of the constraint optimizing problem (3.8) is a vector with length of H_p. The first element of the solution is employed as control output. The pulse width modulation transforms the control output of the power to the duty cycle of the power signal. MPC-QUAN rounds off the control output, aforementioned as the final output while MPC-PWM employs a PWM mechanism described in the previous section to approximate the control output.

The baseline Reactive (Reactive Thermal Control) is a modified version of reactive speed control of embedded real-time systems [95]. The key design point of Reactive is that whenever the thermal threshold is hit, the frequency corresponding to equilibrium temperature (thermal bound in our case) is applied. Otherwise, the highest available frequency is applied. The original version of reactive speed control works at the level of tasks, that is, the frequency changes during the duration of one task running. Reactive, however, only changes frequency at the end of a sampling period. If all the parameters, both power and thermal related, are accurate, Reactive can enforce the thermal threshold effectively. However if there are uncertainties of parameters, the equilibrium temperature cannot precisely enforce the temperature bound.

Constant Power Variation

This set of simulations is designed to evaluate the performance of RT-MTC when there is constant deviation between the estimated and the real tasks power. In these simulations, we compare RT-MTC to the other baselines when the power ratio of all tasks running on the target multicore processor is 4.0, that is, the real power of the tasks is 4 times that of the estimated power. The value of power ratio is chosen intentionally to show the capability of RT-MTC to counteract heavy disturbances, a major benefit of control-theoretic thermal

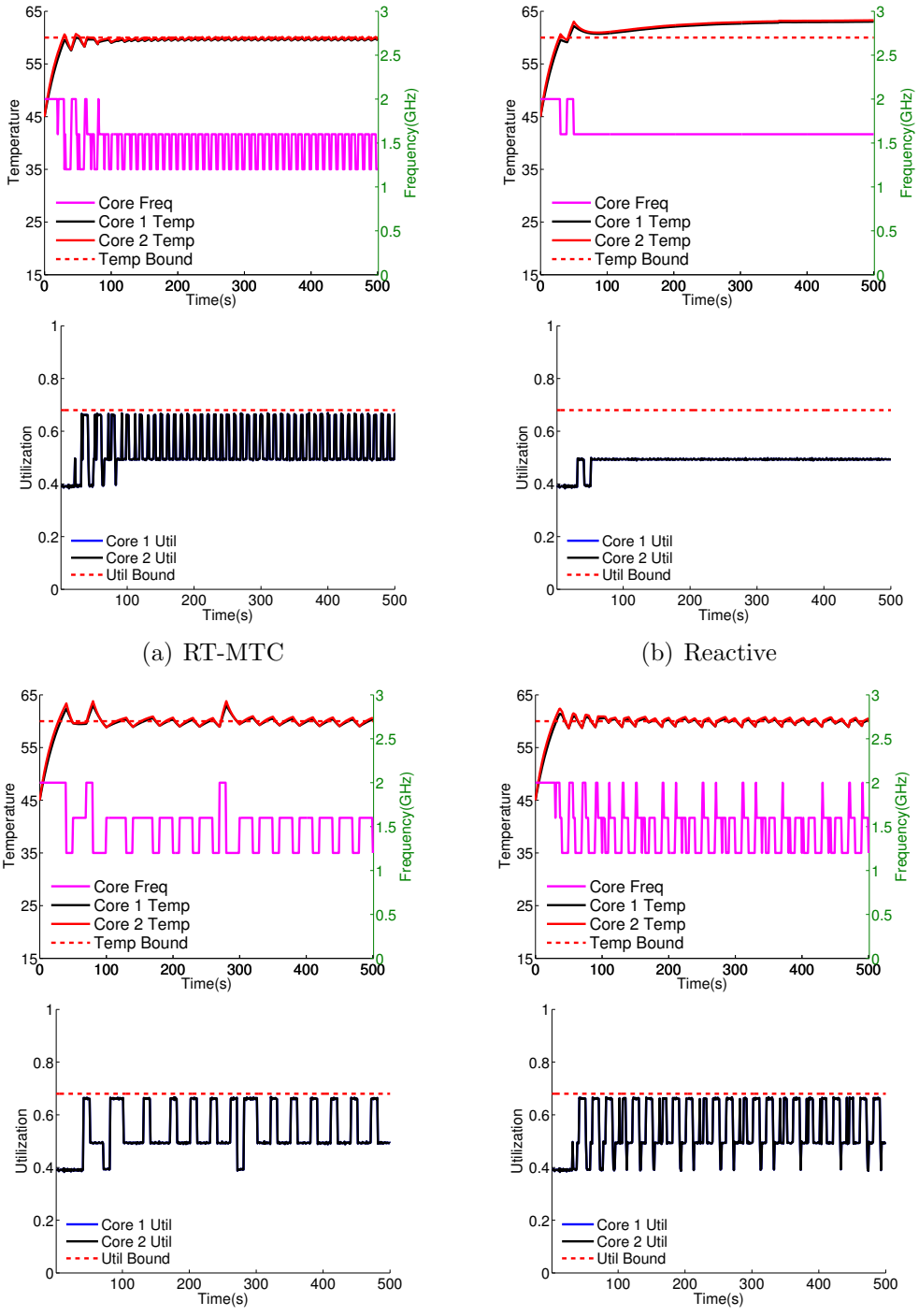
control. In this simulation, we expect RT-MTC to work resiliently under constant power variation.

Fig. 3.4 compares the performance of RT-MTC, Reactive, MPC-QUAN, and MPC-PWM when the power ratio is 4. We exclude OPEN from the comparison intentionally because it violates the thermal bound during the experiment. Without thermal management, the processor cannot handle the thermal bound violation, and the steady temperature of the two cores reaches $84°C$; this significantly exceeds the $60°C$ temperature threshold and likely to trigger the internal hardware thermal control.

As shown in the top figure in Fig. 3.4(a), the temperature under RT-MTC converges to the temperature set point $60°C$. The slight oscillation in converged temperature, which can be seen in Fig. 3.4(d), is caused by the sampling period. If the temperature surpasses the bound within the sampling period ($10s$ in this experiment) RT-MTC cannot respond to enforce the thermal bound. Meanwhile, we also observe the frequency switches between 3 levels guided by PWM according to RT-MTC's output.

The bottom half of Fig. 3.4(a) shows the utilization of the multicore processor. As seen in the figure, the utilization is always below the utilization bound, validating that RT-MTC can enforce the real-time utilization bound. Because of RT-MTC saturation component, the frequency never switches to the lowest level, which confines utilization under the real-time bound.

Fig. 3.4(b) illustrates the simulation results under Reactive. After two frequency switches, Reactive forces the frequency to stay at $1.6GHz$ even though the temperature violates the thermal bound. Recall the algorithm of Reactive: if the thermal bound is hit, the frequency will change to the predefined level to enforce the equilibrium temperature, which, otherwise, is calculated based on the nominal model. In this case, the predefined frequency level is $1.6GHz$. However, in this simulation, the power ratio is 4.0 rather than 1.0 used by Reactive. Hence, at the same frequency, more power is generated and the predefined frequency level in Reactive cannot prohibit the temperature from surpassing the bound. This experiment shows clearly that Reactive is not able to handle thermal management accurately under power uncertainty.



(a) RT-MTC

(b) Reactive

(c) MPC-QUAN

(d) MPC-PWM

Figure 3.4: Constant power variation when power ratio is 4.

Compared to Reactive, RT-MTC follows the temperature set point more precisely under power uncertainty. When the power generated by the processor is overestimated, the processor runs at higher frequency in RT-MTC than Reactive, so that throughput of the systems is improved. When the power is underestimated, likewise, RT-MTC adjusts the processor frequency to consume less power than Reactive, which can not only save power consumption of the workload but also reduce power consumed by the cooling system. Moreover, in this case, Reactive is more likely to trigger internal thermal throttling.

Fig. 3.4(c) and 3.4(d) show the simulation results of MPC-QUAN and MPC-PWM. Both baselines can ensure the temperature set point. However, there is oscillation in both cases. For MPC-QUAN, because of the effect of quantization, the temperature frequently violates the bound slightly. Although MPC-PWM can alleviate the effect of quantization by PWM, the sampling period that we analyzed in RT-MTC also induce oscillation around the thermal bound. Moreover, since MPC works on the margin of constraints, it behaves in a complex, nonlinear way. That makes the oscillation of MPC-PWM greater than that of RT-MTC. On the other hand, MPC can handle effectively the real-time constraints embedded in the constrain optimizing problem (3.8), which then enforces the real-time constraints, that is, the utilization bound.

The major advantage of RT-MTC over MPC-like methods is the reduction of running overhead and implementation complexity. When employing MPC, the controller must solve online the constrained optimization problem, which is notably computation intensive [62]. In contrast, RT-MTC only involves computation of a linear function. Moreover, although there are a few of commercial or open source optimization solver, porting them to solve MPC is still a difficult task.

Dynamic Power Variation

This set of simulations is designed to evaluate the case when the power ratio of tasks deviate from the estimation dynamically. Since tasks often experience different stages of processing, the power of tasks changes frequently. Thus, dynamic power variation is a common source of uncertainty for thermal management. In this simulation, we also assume asymmetric power ratio variation: that is, cores consuming different power when running. For the simulations

in this section, we assume the power ratio of Core 1 rises to 4.0 at $200s$ and then decreases to 0.5 at $300s$ while Core 2 keeps the power unchanged.

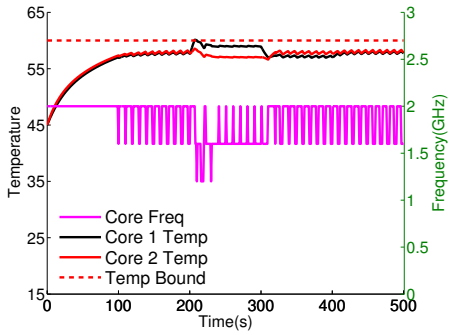
Similarly to the case of constant power variation, OPEN violates the thermal bound under dynamic power variation. However, since only the power of core 1 increases, the temperature of both cores rises less than if the power of both cores varied.

Fig. 3.5 shows the simulation results of different algorithms under dynamic power variation. Fig 3.5(a) shows that the temperature of core 1 is below the temperature bound under RT-MTC, validating that RT-MTC is able to ensure the thermal bound under dynamic power variation. We observe that RT-MTC responds to the abrupt temperature increase from $200s$ to $300s$. So when power decreases, the temperature is still able to stay near the temperature bound.

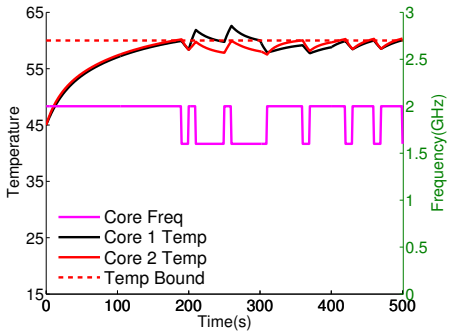
Unlike the previous experiments, Reactive has no steady temperature error in the simulation, as shown in Fig. 3.5(b). As only one core's power rises, the heat generated by the processor is less than that when both cores' power rise; hence the predefined frequency level can enforce the thermal bound. However, we observes spikes in temperature which violates the thermal bound. These spikes occur because the reactive mechanism only responds to thermal violation passively, compared to RT-MTC where the feedback controller is designed intentionally to accommodate a temperature variation so as to offset thermal violation.

Fig. 3.5(c) and 3.5(d) show the results under MPC-QUAN and MPC-PWM, respectively. When subjected to dynamic power variation, both MPC baselines can keep the temperature around the thermal bound. But similarly to the case of constant power variation, quantization and nonlinear control behavior cause oscillation.

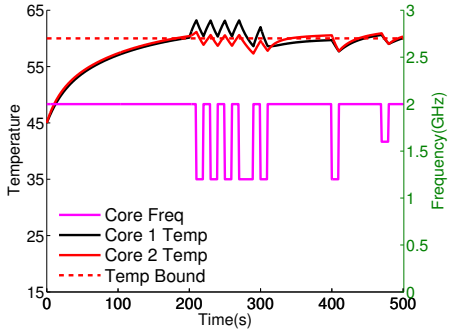
To explore the limits of robustness of RT-MTC, we also perform additional simulation experiments under wider uncertainty than the two simulations discussed here. The results also indicate that RT-MTC is more robust than other algorithms when subjected to uncertainties. More details on these experiments may be found in [29].



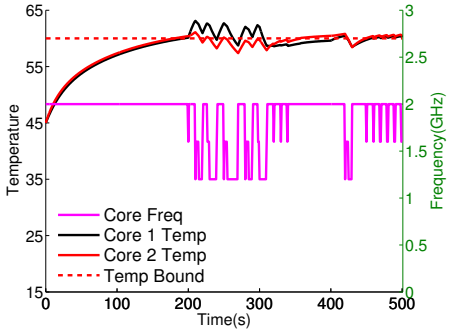
(a) RT-MTC



(b) Reactive



(c) MPC-QUAN



(d) MPC-PWM

Figure 3.5: Dynamic power variation

3.8 Related Work

There has been significant work on thermal aware real-time scheduling for both single-core processors [14, 95] and multicore processors [13, 15, 24]. Those algorithms rely on accurate models about the thermal characteristics of the processors, and hence cannot effectively deal with uncertainties in thermal characteristics such as power consumption and ambient temperature. Moreover, they usually require fine-grained scheduling decisions that require kernel-level implementations. In contrast, our feedback control approach is implemented in user space without modifications to the kernel and therefore can be easily deployed in existing systems.

Control-theoretic thermal management has been explored for non-real-time systems. Donald and Martonosi present a general framework of dynamic thermal management for multicore processors [20]. Essentially, the proposed framework is a hierarchical feedback control loop with PI controllers, but it does not provide real-time performance guarantees. Several papers [71,99,104–106] have adopted model predictive control or online convex optimization for dynamic thermal management. None of these works is concerned with maintaining real-time performance. In addition, control approaches based on model predictive control and convex optimization has higher computation complexity than our efficient proportional control approach. Moreover, our approach deals with *discrete* voltage/frequency levels, a practical issue associated with DVFS which is ignored by the aforementioned control solutions [71,105,106].

Control-theoretic approaches have recently been proposed for thermal management of real-time systems [28, 54]. Our previous work [28] proposed a feedback control algorithm that enforces thermal and real-time constraints simultaneously. That work adjusts the rate of periodic real-time tasks as the control knob, whereas RT-MTC employs DVFS that does not require applications to support variable rates. Lindberg [54] proposed a feedback control framework to manage both temperature and media performance. Both algorithms [28, 54] are designed for single-core processors and cannot deal with multicore processors as they are not cognizant of inter-core thermal coupling in multicore processors.

Different from prior research handle thermal management on hardware level [11, 40, 81, 82], RT-MTC mainly focus on system level thermal management of multicore processors. Two

aspects differentiate hardware and system level thermal management. First, thermal dynamics on hardware level is faster, with time constant at milliseconds [11]. In contrast at system level thermal dynamics of the processor is relative slow and with time constant in seconds [37]. Second, hardware thermal management usually adopt low level control knobs, e.g., clock gating or pipeline throttling, which can not be exposed as system level interfaces. In contrast, system level thermal management employs high-level knobs, e.g., DVFS, that are supported by most operating systems.

3.9 Summary

Embedded real-time systems face significant challenges in thermal management with their adoption of multicore processors of increasing power density. Such systems require the temperatures and real-time performance of *multiple* cores to be controlled simultaneously, leading to multi-input-multi-output control problems with inter-core thermal coupling. This paper presents *Real-Time Multicore Thermal Control (RT-MTC)*, the first feedback thermal control algorithm specifically designed for multicore embedded real-time systems. RT-MTC dynamically enforces both the temperature and the CPU utilization bounds of a multicore processor through DVFS. The strength of RT-MTC lies in both its control-theoretic approach and its practical design. RT-MTC employs a highly efficient controller that integrates saturation and proportional control components rigorously designed to enforce the desired core temperature and CPU utilization bounds. Moreover, It handles discrete frequencies through Pulse Width Modulation (PWM) that enables RT-MTC to achieve effective thermal control with only a small number of frequencies typical in current processors. The robustness and advantages of RT-MTC over existing thermal control approaches are demonstrated through extensive simulations under a wide range of power consumptions.

Chapter 4

Robust Control-theoretic Thermal Balancing for Server Clusters

4.1 Introduction

Unlike thermal throttling on individual processor, *thermal balancing* aims to balance the temperatures of different processors through dynamic load distrbution in a server cluster. Thermal balancing is an attractive approach to thermal management in server clusters for three imporant reasons. First, thermal balancing can effectively mitigates hotspots in a cluster and hence effectively reduce the cooling cost for data centers. For example, a study of a real data center showed that reducing the temperature difference from $10°C$ to $2°C$ could result in close to a 25% reduction in total energy costs associated with the cooling infrastructure [67]. Second, in contrast to other thermal management mechanisms such as throttling and dynamic voltage scaling, thermal balancing can prevent server overheating without causing performance degradation. Finally, thermal balancing can be applied to heterogeneous server clusters including legacy processors that do not support throttling or dynamic voltage scaling.

To implement thermal balancing, a thermal balancer may be implemented on the gateway of a server cluster that provide a same set of services on multiple servers. The thermal balancer intercepts service requests from clients and then dynamically forwards them to appropriate servers based on online temperature measurement. While thermal balancing shares similarities with load balancing, it faces several unique challenges. First, while a load balancer is designed to balance the *load* on different servers, a thermal balancer aims to balance the

temperatures of different servers. While the temperature of a server is related to its load, the thermal dynamics of a server are inherently more complex than system load, because the temperature not only depends on the current load but also its history. It is therefore important to incorporate the thermal dynamics in the design of the thermal balancing algorithm. Finally, thermal balancing must handle uncertain thermal characteristics, such as varying power consumption, thermal faults, and varying ambient temperature.

To tackle these challenges, we present *Control-theoretic Thermal Balancing (CTB)*, a novel thermal balancing approach based on a control-theoretic underpinning. CTB employs a feedback control loop that periodically monitors the temperature and CPU utilization of different servers in a cluster, and redistributes clients' service requests among different processors to dynamically balance their temperature. CTB features control algorithms that are rigorously designed and analyzed based on optimal control theory. Specifically, this paper makes the following main contributions.

- We derive a difference equation model that characterizes the thermal dynamics of server clusters as a foundation for control designs and analysis of thermal balancing.

- We present the design and stability analysis of two CTB algorithms analytically designed based on optimal control theory. CTB-T uses processor temperature as feedback while CTB-UT uses both processor temperature and CPU utilization to significantly reduce task reallocation cost.

- We provide simulation results that demonstrate CTB algorithms can deliver robust thermal balancing in face of a wide range of uncertainties, including different power consumption incurred by different tasks, different ambient temperatures of different servers, and thermal faults.

In the rest of this chapter, Section 4.2 formulates the thermal balancing problem from a control perspective. Section 4.3 details the design of CTB algorithms. Section 4.4 provides the simulation results. Section 4.5 introduces related works. Section 4.6 summarizes this chapter.

4.2 Problem Formulation

In this section we first describe the system and thermal models and then formulate the thermal balancing problem.

4.2.1 System model

A cluster consists of n homogeneous single-processor servers $\{S_i|1 \leq i \leq n\}$ connected by networks. All servers host a same set of services. A client may periodically invoke a service hosted by a server, where the periodic processing of the request corresponds to a periodic task T_i on the server. Let the execution time and the period of T_i are c_i and p_i, respectively. Then the (CPU) utilization of T_i is $U_i = \frac{c_i}{p_i}$.

As managing the temperature of processors is a major concern in server cluster, we focus on the thermal and power properties of processors. Extending our work to thermal control for the other system components is part of our future work. The processor of each server has *estimated* active power P_a (e.g., the active power in the specification of the processor) when it is executing tasks. It is important to note that the *actual* active power of a processor may deviate from the estimated at run time and different tasks may incur different power consumption [11, 45]. For example, an earlier study showed that the active power consumption of different applications may differ by as much as 35% [11]. In earlier literature some researchers [46] referred to such significant power variation during run time as *power phase* behavior. At the instruction level, different instruction types, inter-instruction overhead, memory system state and pipeline related effects cause fluctuation of task powers [85]. When the processor is idle, the processor switches to a low power mode and consumes power of P_{idle}.

We employ a widely adopted thermal model [5, 41, 81] for the processor Pr_i as follows.

$$\frac{dT_i(t)}{dt} = -c_{i,2}(T_i(t) - T_0) + c_{i,1}P_i(t) \tag{4.1}$$

where $T_i'(t)$ is the temperature of processor Pr_i, $c_{i,1}, c_{i,2}$ are the constant pertained to the thermal characteristics of the processor and T_0 is ambient temperature. Let $T_i'(t) = T_i(t) - T_0$.

We can write equation (4.1) in a more compact form

$$\frac{dT_i'(t)}{dt} = -c_{i,2}T_i'(t) + c_{i,1}P_i(t).$$

For the whole system with n homogeneous processors, the thermal model is the aggregation of the individual processor's thermal model,

$$\dot{\mathbf{T}}'(t) = \mathbf{A}\mathbf{T}'(t) + \mathbf{B}\mathbf{P}(t)$$

where

$$\mathbf{T}'(t) = [T_1'(t), T_2'(t), ..., T_n'(t)]^T$$

and

$$\mathbf{P}(t) = [P_1(t), P_2(t), ..., P_n(t)]^T$$

while the constant matrices are

$$\mathbf{A}=\begin{bmatrix} -c_{1,2} & & 0 \\ & \ddots & \\ 0 & & -c_{n,2} \end{bmatrix}, \mathbf{B}=\begin{bmatrix} c_{1,1} & & 0 \\ & \ddots & \\ 0 & & c_{n,1} \end{bmatrix}.$$

Thus, the thermal model of the cluster can be written as

$$\frac{d\mathbf{T}'(t)}{dt} = \mathbf{A}\mathbf{T}'(t) + \mathbf{B}\mathbf{P}(t) \tag{4.2}$$

Our thermal model does not consider the correlation of the temperatures of different servers based on recent studies that showed the thermal correlation between severs is insignificant in a commercial server cluster [16].

4.2.2 Dynamic Model for Thermal Balancing

As our CTB algorithms are designed to control the processors' temperatures through load redistribution, we need to establish a difference equation model to characterize their dynamic relationship between the processors' temperature and their CPU utilization, where the CPU utilization is the fraction of time when the CPU is running tasks. For the purpose of control

design. As the feedback control loops of CTB are invoked once every sampling period W, we first discretize the continuous thermal model. Let $P(k)$ and $T'(k)$ denote the discretized power and temperature, respectively, which are measured at sampling time kW.

By *bilinear transformation* [26], the continuous form of the thermal model (4.2) is discretized as

$$\mathbf{T'}(k+1) = \mathbf{\Phi T'}(k) + \mathbf{\Gamma' P}(k) \tag{4.3}$$

where $\mathbf{\Phi} = (\mathbf{I} + \frac{\mathbf{AW}}{\mathbf{2}})(\mathbf{I} - \frac{\mathbf{AW}}{\mathbf{2}})^{-1}$ and $\mathbf{\Gamma'} = (\mathbf{I} - \frac{\mathbf{AW}}{\mathbf{2}})^{-\mathbf{1}}\mathbf{B}\sqrt{W}$ and W is the length of sampling interval.

Next we characterize the relationship between the CPU utilization and the power consumption of a processor. Let $U_i(k)$ denote the CPU utilization of the processor Pr_i in the k^{th} sampling period. Then the average power of the processor in k^{th} sampling period, $\bar{P}_i(k)$, can be written as

$$\begin{aligned} \bar{P}_i(k) &= G_i P_a U_i(k) + P_{idle}(1 - U_i(k)) \\ &= G_i(P_a - P_{idle})U_i(k) + P_{idle}. \end{aligned} \tag{4.4}$$

where G_i is the ratio between the average *actual* and *estimated* active power of the processor. Note that G_i is unknown at design time. An important goal of the our work is to design a control algorithm for thermal balancing that can tolerate a wide range of variations in G_i.

For the thermal control analysis we need to derive a discrete-time model to approximate this system. As the thermal-time constant is large, the effects of transients of power consumption within a sampling period is negligible. Therefore, we substitute average power $\bar{\mathbf{P}}(k) = [\bar{P}_1, \bar{P}_2, ..., \bar{P}_n]^T$ with discrete power $\mathbf{P}(k)$ in (4.3). Combining (4.3) and (4.4) and considering all processors in the server cluster, we get the following dynamic model for the server cluster:

$$\mathbf{T'}(k+1) = \mathbf{\Phi T'}(k) + \mathbf{\Gamma' G}(\mathbf{P}_a - \mathbf{P}_{idle})\mathbf{U}(k) + \mathbf{\Gamma' P}_{idle} \tag{4.5}$$

where $\mathbf{G}$ is defined as $diag(G_1, G_2, ..., G_n)$, $\mathbf{P}_a = [P_a, P_a, ..., P_a]^T$ and $\mathbf{P}_{idle} = [P_{idle}, P_{idle}, ..., P_{idle}]^T$. Let $\mathbf{T}(k) = \mathbf{T'}(k) - \tilde{\mathbf{T}}$, where $\tilde{\mathbf{T}} = \mathbf{\Gamma' P}_{idle}(\mathbf{\Phi} - \mathbf{I})^{-1}$, the thermal model of the system can be rewritten as

$$\mathbf{T}(k+1) = \mathbf{\Phi T}(k) + \mathbf{\Gamma U}(k) \tag{4.6}$$

where $\mathbf{\Gamma} = \mathbf{G}(\mathbf{P_a} - \mathbf{P_{idle}})\mathbf{\Gamma}'$.

It is noted that our model ignores the discrete nature of task utilizations and thus more suitable for servers with a large number tasks each consuming a small fraction of the CPU cycles. This liquid model is a reasonable approximation of many high-performance servers. Extending our work to deal with servers with non-negligible discrete utilization changes is part of our future work.

4.2.3 Thermal Balancing Objective

The objective of thermal balancing is,

$$\min_{k\to\infty} \sum_{i=1}^{n} (T_i(k) - \bar{T}(k))^2 + \rho \sum_{i=1}^{n} \Delta U_i^2(k) \tag{4.7}$$

where $\bar{T}(k)$ is the average temperature, defined as $\bar{T}(k) = \frac{\sum_{i=1}^{n} T_i(k)}{n}$, $\Delta U_i(k)$ is the change to the utilization of processor Pr_i, i.e., the difference between the total utilization of the tasks moved to processor Pr_i and that of the tasks moved from processor Pr_i.

The first term of the objective function aims to reduce the differences among the temperatures of different processors. The second term of the objective function aims at reducing control cost, *i.e.*, the number of tasks redirected. This is important because redirecting a tasks can incur non-negligible performance penalty and overhead , e.g., due to loss of cache states or reestablishing the HTTP session.

4.3 CTB Design and Analysis

In this section we first provide an overview of the *Control-theoretic Thermal Balancing (CTB)* approach. We then present the difference equation model that characterizes the thermal dynamics of a servers cluster. Based on the dynamic model we detail the design and stability analysis of two CTB algorithms.

4.3.1 Overview of CTB

CTB uses online feedback for thermal balancing. A natural choice of feedback is temperature (the controlled variable). However, as temperature responds slowly to load redistribution, thermal balancing solely based on temperature feedback may not be able to regulate temperature quickly. To deal with the problem, the thermal balancer may also employ CPU utilization as feedback. In this work we develop two control algorithms for thermal balancing. *CTB-T* only uses the temperature as feedback while *CTB-UT* employs both utilization and temperature as feedback.

As shown in Figure 4.1, CTB employs a distributed feedback control loop consisting of a *controller* and a *balancer* on the gateway for the cluster, and *monitors* located on the servers. For the CTB-T algorithm, the controller input is a vector, $\mathbf{T}(\mathbf{k})$, which includes the temperature of each processor at the end of the kth sampling period. The output of the controller is a vector of *utilization change*, $\mathbf{\Delta U}(\mathbf{k})$, which indicates the requested change to each processor's utilization. According to the output of the controller, the *balancer* computes the clients' service requests invocations that should be redirected among different servers in the following sample period.

Specifically, CTB-T algorithm works as follows. At the end of the kth sampling period, the feedback loop is invoked and executes the following steps:

1. Each temperature *monitor* sends the temperature in the end of the last sampling period to the controller. Most modern processors integrates on-chip temperature sensors. Alternatively, software techniques based on event counters can be used to estimate processor temperature [5].

2. The *controller* calculates the change to the CPU utilization of every processor, $\mathbf{\Delta U}(\mathbf{k})$, based on the temperature vector $\mathbf{T}(\mathbf{k})$. Then $\mathbf{\Delta U}(k)$ is sent to centralized *balancer*.

3. The *balancer* reallocates tasks among different servers to accommodate the requested utilization change $\mathbf{\Delta U}(k)$. The Balancer first divides all processors into three sets, the *receivers*, the *senders*, and the *neutral* according to the controller output $\Delta U(k)$. The processors in the receivers set have positive utilization changes; the processors in the senders set have negative utilization changes; and the processors in the neutral

set have zero utilization change and hence are not involved in task reallocation in the following sampling period. The Balancer then reallocate tasks from processors in the senders set to processors in the receivers set to accommodate the requested utilization changes specified in $\Delta U(k)$. In the following sampling period, the Balancer directs clients' service invocations to appropriate servers according to the new task allocation.

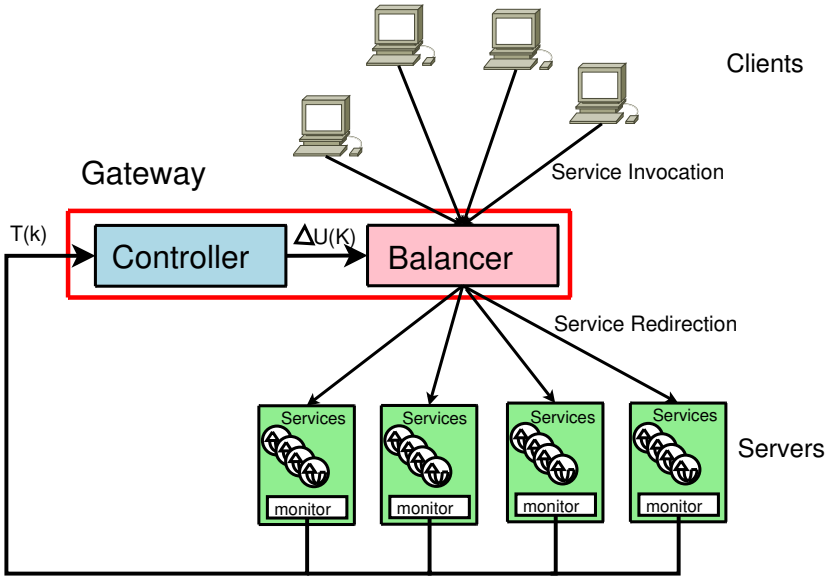


Figure 4.1: The feedback control loop of CTB-T

CTB-UT works in the same way as CTB-T except that it employs a different control algorithm that utilizes both the CPU utilization vector $\mathbf{U}(k)$ and the temperature vector $\mathbf{T}(k)$ as control inputs. Each processor hence runs a utilization monitor in addition to the temperature monitor. The utilization monitor measures the CPU utilization in the last sampling period and sends it to the controller. For example, in Linux, the utilization monitor can use `/proc/stat`file to estimate the CPU utilization in each sampling period. The `/proc/stat`file records the number of *jiffies* since the system start time, when the CPU is in user mode, user mode with low priority (nice), system mode, and when used by the idle tasks. At the end of each sampling period, the utilization monitor reads the counters, and estimates CPU utilization by dividing the number of *jiffies* used by the idle tasks in the last sampling period by the total number of *jiffies* in the same period [97].

4.3.2 Control Design of CTB-T

The core of the CTB algrithm is the controller. Recall the control design has two objectives: (1) to reduce the differences among the temperature of different processors, and (2) to reduce the number of reallocated tasks to reduce overhead. The first goal aims at thermal balancing, while the second goal aims at reducing the control cost. To address both objectives, we choose to design a Linear Quadratic Regulator (LQR) based on optimal control theory. The LQR controller is designed for the following optimization objective.

$$\min_{k\to\infty} \sum_{k} [\mathbf{X}(k)^T \mathbf{Q} \mathbf{X}(k)] + \Delta\rho \mathbf{U}(k)^T \mathbf{\Delta U}(k)]. \tag{4.8}$$

where $\mathbf{Q}$ are weight matrix respectively and $\mathbf{\Delta X}(k)$ is the state of the thermal model (4.6), *i.e.*, $\mathbf{X}(k) = \mathbf{T}(k)$. If we denote

$$\mathbf{L} = \begin{bmatrix} 1-\frac{1}{n} & -\frac{1}{n} & \dots & -\frac{1}{n} \\ -\frac{1}{n} & 1-\frac{1}{n} & \dots & -\frac{1}{n} \\ \vdots & \ddots & \dots & -\frac{1}{n} \\ -\frac{1}{n} & \dots & \dots & 1-\frac{1}{n} \end{bmatrix}$$

and then $\mathbf{Q} = \mathbf{L^T L}$, the thermal balancing objective (4.7) can be transformed to the LQR controller with objectives 4.8. In fact because the difference between processor Pr_i's temperature and the average temperature of all processors is

$$\begin{aligned} \Delta T_i(t) &= T_i(t) - \frac{\sum_{1<k<n} T_k(t)}{n} \\ &= (1-\frac{1}{n})T_i(t) - \frac{1}{n}T_1(t) \dots - \frac{1}{n}T_n(t) \end{aligned} \tag{4.9}$$

The optimization objectives of our LQR controller 4.8 clearly match the thermal balancing and the control cost objectives 4.7.

4.3.3 Control Design of CTB-UT

There is a significant delay between the change in the CPU utilization and the resultant change in the processor temperature due to slow thermal dynamics. To improve the responsiveness of thermal balancer, CTB-UT employs both temperature and utilization as feedback for thermal balancing. CTB-UT also employs an LQR controller, but its state variables include temperatures and utilization.

The dynamics of utilization can be modeled as [61]

$$\mathbf{U}(k+1) = \mathbf{U}(k) + \mathbf{\Delta U}(k). \tag{4.10}$$

Combining 4.10 and thermal model (4.6), we can model the relationship between the temperatures and the utilization

$$\begin{aligned} \begin{bmatrix} \mathbf{T}(k+1) \\ \mathbf{U}(k+1) \end{bmatrix} &= \begin{bmatrix} \Phi & \Gamma \\ 0 & I \end{bmatrix} \begin{bmatrix} \mathbf{T}(k) \\ \mathbf{U}(k) \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \mathbf{\Delta U}(k) \\ &= \mathbf{\Phi_{UT}} \begin{bmatrix} \mathbf{T}(k) \\ \mathbf{U}(k) \end{bmatrix} + \mathbf{\Gamma_{UT}} \mathbf{\Delta U}(k) \end{aligned} \tag{4.11}$$

The LQR controller of CTB-UT has the form as follows

$$\mathbf{\Delta U}(k) = - \begin{bmatrix} \mathbf{K_T} & \mathbf{K_U} \end{bmatrix} \begin{bmatrix} \mathbf{T}(k) \\ \mathbf{U}(k) \end{bmatrix} \tag{4.12}$$

The optimization objective of the LQR controller of CTB-UT is:

$$\min_{k \to \infty} \sum_{k} \left\{ \begin{bmatrix} \mathbf{T}(k) \\ \mathbf{U}(k) \end{bmatrix}^T \mathbf{Q_{UT}} \begin{bmatrix} \mathbf{T}(k) \\ \mathbf{U}(k) \end{bmatrix} + \mathbf{\Delta U}^T(k) \mathbf{R_{UT}} \mathbf{\Delta U}(k) \right\} \tag{4.13}$$

where $\mathbf{Q_{UT}} = \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{0} & \lambda \end{bmatrix}^T \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{0} & \lambda \end{bmatrix}$, $\mathbf{R_{UT}} = \rho \mathbf{I}$ and λ is a tunable parameter. The first term of optimization objective is responsible for balancing both temperature and utilization. Through tuning λ, we can adjust the weight between temperature balancing and utilization balancing. If $\lambda < 1$, the controller is biased for thermal balancing. If $\lambda > 1$ the controller

is biased for load balancing which responds quickly to load change. With an appropriate λ, CTB-UT can combine the benefits of thermal balancing and load balancing.

4.3.4 Stability and Robustness

The stability of a server cluster with CBT is defined as convergence of the temperatures of all processors to their average temperatures. We design the LQR controllers based on a *nominal* system with $G = I$, that is, the power and utilization equals their estimation. It is important to derive the region of G where the system remains stable.

Theorem 4. *The stable range of the* $\mathbf{G}$ *in CTB-T is*

$$diag\left(\frac{1}{1+\sqrt{\alpha}}\right) < \mathbf{G} < diag\left(\frac{1}{1-\sqrt{\alpha}}\right), \tag{4.14}$$

where

$$\alpha = \frac{R}{R + W\Gamma^T S\Gamma}.$$

and W *is the sampling period and* S *is the solution of Algebraic Riccati Equation (ARE) [26],* $S = \Phi^T[S - S\Gamma R^{-1}\Gamma^T S]\Phi + Q$.

The proof of Theorem 4 can be derived directly from Corollary 11.4.1 in [65]. It is noted that since $R = \rho I$ the robust stable region of CTB-T varies with ρ.

Note this approach to robustness analysis is not applicable to CTB-UT. When power gain G varies, Φ_{UT} also changes. Thus we cannot use Theorem 4 to derive the stability region since the ARE does not have a fixed solution like the case of CTB-T. Instead, we develop a numerical solution to calculate the stability region.

First we derive the closed-loop systems based on thermal model of CTB-UT (4.11) and the optimal controller (4.12). The closed-loop system has the form

$$\begin{aligned}\begin{bmatrix} T(k+1) \\ U(k+1) \end{bmatrix} &= (\Phi_{UT} - \begin{bmatrix} K_T & K_U \end{bmatrix} \Gamma_{UT}) \begin{bmatrix} T(k) \\ U(k) \end{bmatrix} \\ &= \Phi_C \begin{bmatrix} T(k) \\ U(k) \end{bmatrix}\end{aligned} \tag{4.15}$$

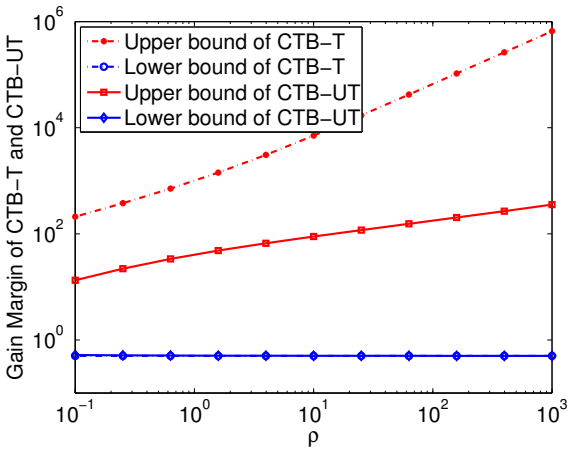


Figure 4.2: Analytical Power Gain Stable Region of CTB-T and CTB-UT

For a fixed ρ, Φ_C characterizes dynamics of the closed-loop system. According to control theory, if the eigenvalues of Φ_C is in the unit circle the dynamical system is stable. We can acquire the stable region of G based on the eigenvalues of Φ_C under different values of G.

Figure 4.2 illustrates the stable region of CTB-T and CTB-UT derived analytically. Both CTB algorithms can maintain stability under a wide range of power gains G. For example, when $\rho = 1$ the stable gain margin range of CTB-T and CTB-UT are $[0.50, 1474.30]$ and $[0.51, 42.91]$ respectively. CTB-UT has a smaller stability region than CTB-T. Therefore the choice between the them should consider the tradeoff between their responsiveness to variations and their robustness with regard to varying power consumptions. Another trends is that the stable region of both CTB-T and CTB-UT increases with ρ, the relative weight of the control cost and the temperature difference in the optimization objective. A higher ρ, leads to lower control cost and hence a higher degree of robustness against high power consumption.

4.4 Evaluation

We evaluate our CTB algorithms using simulations. We first describe the simulation setup and three baseline algorithms for comparison. We then presents simulation results with varying power consumption, thermal faults, and varying ambient temperatures.

4.4.1 Simulation Setup

To evaluate the CTB algorithms we develop an event-driven simulator which simulates a cluster consisting of 16 servers. The CTB algorithms are implemented using the Optimal Control Toolbox of MATLAB.

The task set running on each processor consists of 50 periodic soft real-time tasks. The Eearlies Deadline First (EDF) scheduling algorithm [55] is employed to schedule all these tasks. The period p_i of each task T_i is chosen randomly with a uniform distribution in the range $[900ms, 1100ms]$. The deadline of each task equals its period.

Each processor simulated is a $2.6GHz$ Pentium 4 (P4) processor with $130nm$ Northwood core. All thermal related parameters except thermal capacitance shown in Table 4.1 are based on the Intel technical specification [44]. The thermal capacitance is acquired by simulating P4 on Hotspot [41], an architecture level simulator.

Parameter	Notation	Value
Ambient temperature	T_0	$45°C$
Max case temperature	T_c	$75°C$
Estimated active power	P_a	$51.9W$
Idle power*	P_i	$13.3W$
Thermal capacitance	C_{th}	$295.7J/K$
Thermal resistance	R_{th}	$0.467K/W$

* Enhanced Halt Mode is available [86]

Table 4.1: Power and thermal parameters

4.4.2 Baseline Algorithms

We use three baseline algorithms as baselines for comparison in our simulations: an OPEN-loop algorithm (OPEN), a Load Balancing (LB) algorithm and a Heuristic Thermal Balancing (HTB) algorithm. OPEN does not perform any thermal or load balancing. Tasks are always executed on their initial processors.

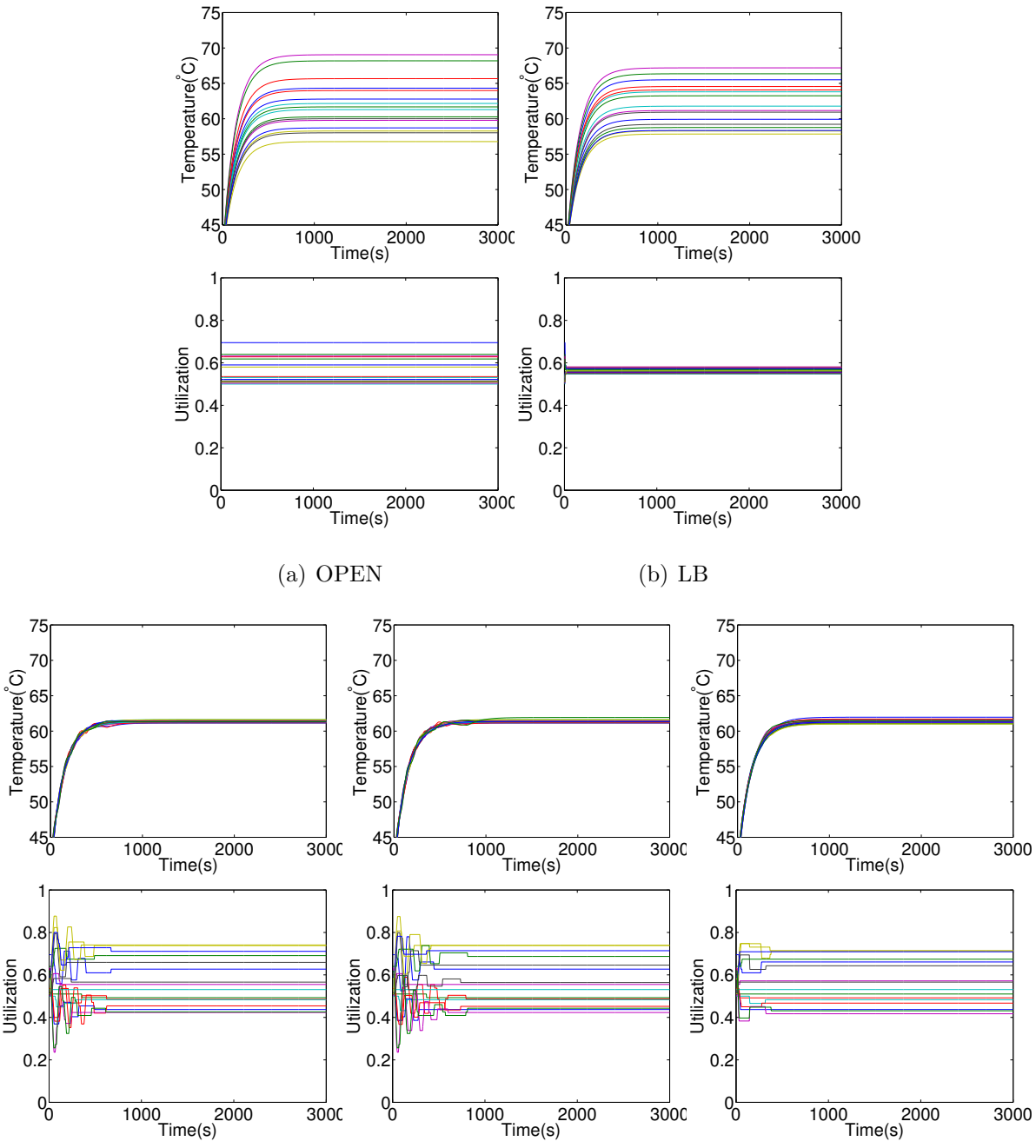
The LB algorithm is designed to dynamically balance the CPU utilization of different processors. The invocation period of the LB algorithm is the same as the sampling period of CTB. In the end of each period, LB measures the utilization of every processor in the last sampling period and then redistributes the tasks to balance the utilization of different processors.

Like the CTB algorithms, the HTB algorithm is designed to balance the temperatures of different processors based on temperature feedback. In contrast to the optimal control approach adopted by CTB, HTB employs a simple heuristic algorithm to reallocate tasks among the processors. The heuristics is based on the observation that the steady-state temperature is proportional to the utilization of the processor. In the end of each sampling period, HTB changes the utilization of a processor proportionally to the difference between its temperature and the average temperature of all processors, i.e., $\delta u_i = f_t * [\frac{T_1}{n} + ...(1 - \frac{T_i}{n})... + \frac{T_n}{n}]$. f_t is the ratio between the steady temperature and the utilization, deriving by setting that $\mathbf{T}(k+1) = \mathbf{T}(k)$ in thermal dynamic equation (4.6). Note that the HTB algorithm has two key differences from the CTB algorithms: (1) it is not designed to reduce the number of tasks redistribution (*i.e.*, control cost); and (2) it ignores the thermal dynamics of the system which may influence the transient response to system variations.

4.4.3 Effect of Thermal Balancing

The first set of simulations evaluates the capability of the CTB algorithms to achieve thermal balance. Each simulation run lasts for 3000s. At starting time, the utilization of each processor is assigned randomly in the range $[0.5, 0.7]$. The power ratio of tasks are randomly selected in the range $[0.6, 1]$. We set the tunable parameter $\rho = 1.0$ for CTB, and $\rho = 1, \lambda = 0.005$ for CTB-UT.

The comparison between CTB and the baseline algorithms are illustrated in Figure 4.3. Since no temperature balancing is applied under OPEN, the maximum temperature difference between processors is $11.8°C$ as shown in Figure 4.3(a). While LB effectively balances the utilization of the processors, the maximum temperature difference is reduced only slightly to $9.9°C$. This results shows that load balancing is not effective in balancing temperature in server clusters due to the varying power consumption among different tasks. In contrast,



(a) OPEN (b) LB

(c) HTB (d) CTB-T (e) CTB-UT

Figure 4.3: Temperatures and CPU utilization of all processors under CTB and baseline algorithms

the feedback-based thermal balancing algorithms, CTB-T, CTB-UT and HTB, effectively balance the temperatures reducing the temperature difference within $9.9°C$ in steady states.

The comparison between LB and the thermal balancing algorithms indicates inherent tradeoff between performance and temperature in server clusters. LB results in more balanced CPU utilization among processors which may lead to higher average performance. The thermal balancing algorithms, on the other hand, results in more balanced temperature at the cost of different CPU utilization among servers. Thermal management has become increasingly important due to the extremely high cooling cost in data centers today [67]. Note the reduction of the maximum temperature difference from $9.9°C$ (under LB) to $0.2°C$ (under CTB algorithms) can have a significant impact on the cooling cost of server clusters. For example, previous work showed that reducing the temperature difference from $10.0°C$ to $2°C$ will result in close to a 25% reduction in total energy costs associated with the cooling infrastructure [67].

4.4.4 Comparison of Thermal Balancing Algorithms

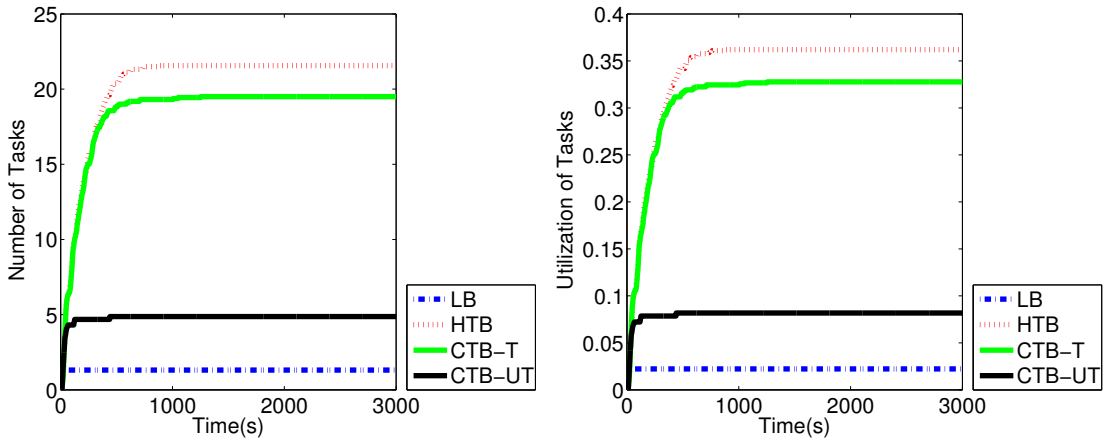
While HTB and both CTB algorithms balance temperatures effectively, there are two important differences in their performance. First, the CTB algorithms, particularly CTB-UT, converges to a steady state with balanced temperature significantly faster than HTB. Table 4.2 compares the *convergence time*, *i.e.*, the time from beginning of the run to the time instant when the last task reallocation is completed. The convergence time of CTB-UT is only one quarter of CTB-T and one sixth of HTB. This result shows that CTB-UT is the most responsive to system variation. As discussed earlier, the responsiveness of CTB-UT results from its design that uses utilization in addition to temperature as feedback.

Algorithm Type	OPEN	LB	HTB	CTB-T	CTB-UT
Converge Time(s)	N/A	10	1540	870	260
Max Temperature Difference($°C$)	11.9	9.9	0.2	0.2	0.2

Table 4.2: Comparison of different algorithms

Second, the overhead of CTB-UT is also lower than HTB and CTB-T. As shown in Figure 4.4, on average HTB and CTB-T reallocated 7.8 and 11.0 tasks, respectively, per processor

before converging to steady states. In comparison, CTB-UT converges to the steady state after reallocating only 5 tasks per processor. This result demonstrates the effectiveness of incorporating control cost in the optimal control design, especially when combined with both utilization and temperature as feedbacks.



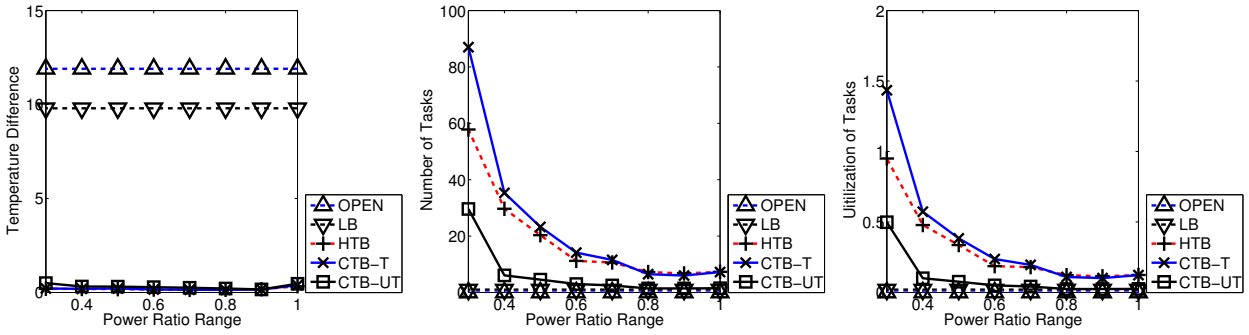
(a) Reallocated tasks per processor

(b) Average total utilization of reallocated tasks per processor

Figure 4.4: Comparison of overhead due to tasks reallocation

To compare the algorithms with a wide range of power consumption, we rerun the simulation with different power distributions for the task set. As shown in Figure 4.5, all the thermal balancing algorithms (HTB, CTB-T, and CTB-UT) effectively maintains temperature balance under the wide range of power ratios used in the simulations, while LB and OPEN result in significant differences in temperatures. Moreover, CTB-UT consistently outperforms CTB-T and HTB in term of reallocation cost.

Figure 4.5 shows the temperature difference and overhead of different algorithms when power distribution of tasks changed. The horizontal axis of the figure is the lower bound of power range, for example, 0.8 means the power ratio of tasks distributes in the range $[0.8, 1]$. In all power distribution, CTB achieves the equivalent temperature difference of HTB and has significant overhead reduction.



(a) Temperature difference (b) Average number of reallocated tasks (c) Average utilization of reallocated tasks

Figure 4.5: Comparison of algorithms with different range of power ratio. The x axis represents the lower bound of the power ratio. For each data point shown in this figure, the power ratio of tasks are randomly chosen in the range $[x, 1]$.

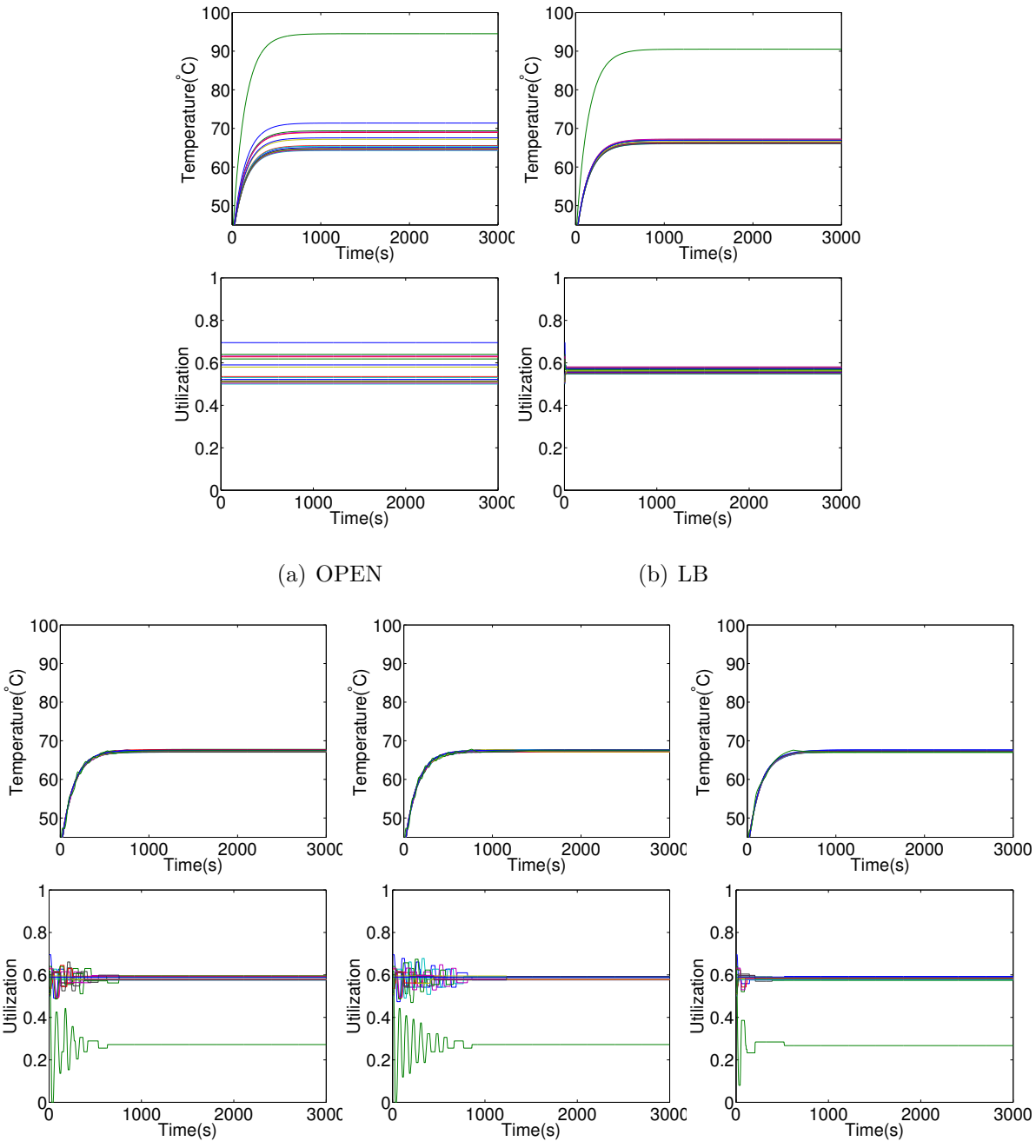
4.4.5 Thermal Fault

This set of simulations test the system's capability to handle thermal faults in servers such as the failure of their cooling system. In this case a robust thermal balancing algorithm should dynamically reallocate tasks from the servers with thermal faults to other servers to maintain thermal balance in the cluster when possible. We simulate the failure of a fan in a server by doubling its thermal resistance, R_{th} [23], in the beginning of the run.

Figure 4.6 shows the simulation results. As expected, OPEN and LB cannot deal with the temperature increase in the server with thermal fault. In contrast, the thermal balancing algorithms effectively maintain the thermal balance by reducing the utilization of the server with thermal fault. This result demonstrates the robustness of the feedback-based thermal balancing algorithms. As shown in Figures 4.7, CTB-UT induces significantly lower cost than CTB-T and HTB for task reallocation.

4.4.6 Robustness against Different Ambient Temperatures

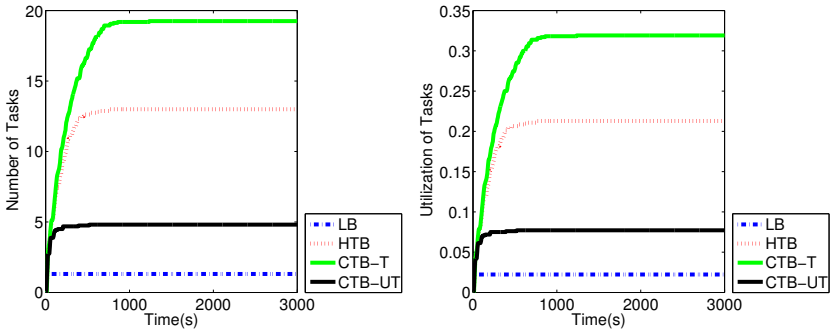
We now evaluate the algorithms' capability to handle the case where different processors experience varying ambient temperatures. In each run the difference of ambient temperature of processors is distributed uniformly in the rang $[40°C, 50°C]$



(a) OPEN (b) LB

(c) HTB (d) CTB-T (e) CTB-UT

Figure 4.6: Temperatures and CPU utilization of all processors under CTB and baseline algorithms



(a) Reallocated Tasks per Processor

(b) Average Total Utilization of Reallocated Tasks per Processor

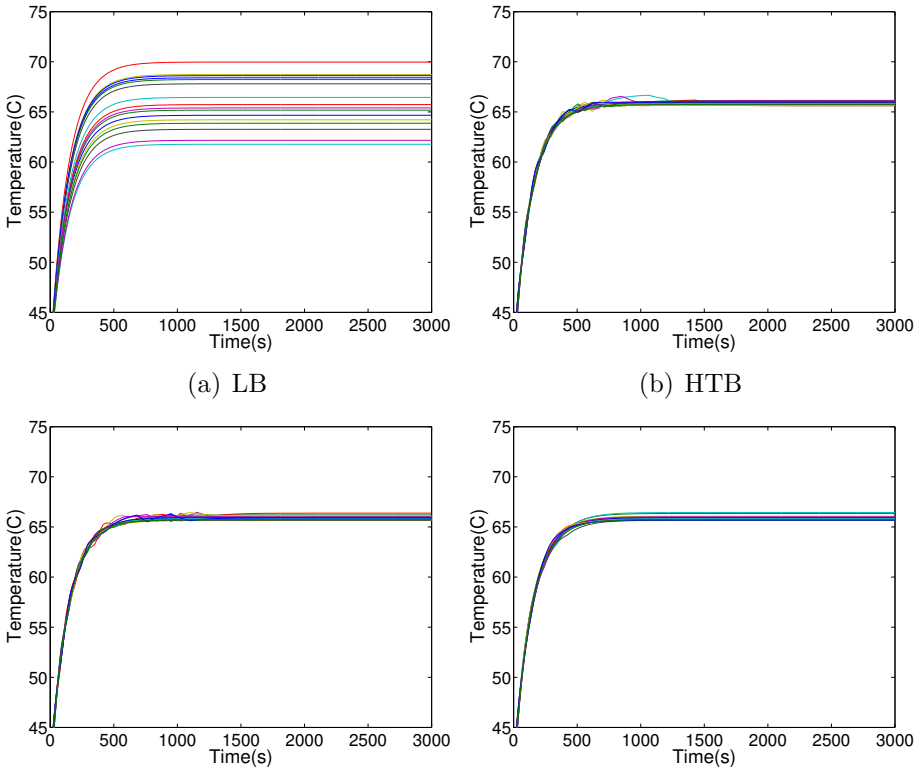
Figure 4.7: Comparison of overhead due to tasks reallocation.

As shown in Figure 4.8, all thermal balancing algorithms effectively balances the temperatures despite varying ambient temperatures among processors, while LB results in significant differences in processor temperatures. Furthermore, CTB-UT incurs the lowest cost in term of task reallocations among the thermal balancing algorithms as shown in Figure 4.9.

To further test the robustness of the algorithms against variations in ambient temperatures, we repeat the simulations with ambient temperatures varying in different ranges. As shown in Figure 4.10, all thermal balancing algorithms maintain thermal balance even when the ambient temperatures od different processors vary by as much as $20°C$. CTB-UT consistently leads to significantly fewer task reallocations than CTB-T and HTB. These results show the robustness and efficiency of CTB-UT under varying ambient temperatures.

4.5 Related Work

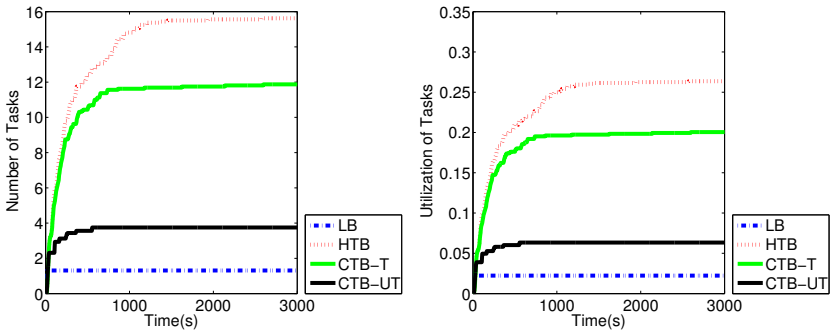
Feedback-based thermal management has been applied at different levels of computer systems. At the architecture level, the authors of [81] employ feedback control to regulate the instruction fetch rate to control the temperature of the processor. Feedback control is also used to manipulate clock gating for mitigating the thermal pressure [11]. Heat-and-Run [31] balances the temperature of different cores in a multicore processor through instruction migration.



(a) LB (b) HTB

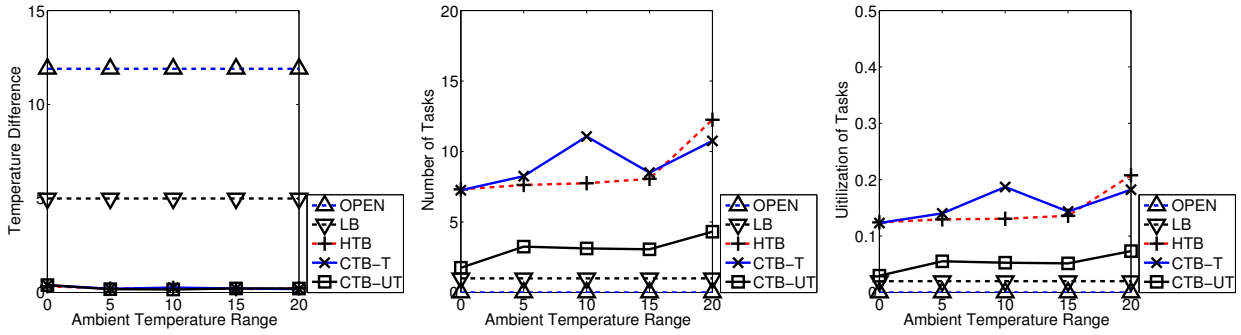
(c) CTB-T (d) CTB-UT

Figure 4.8: Temperatures and CPU utilization of all processors under CTB and baseline algorithms



(a) Reallocated Tasks per Processor (b) Average Total Utilization of Reallocated Tasks per Processor

Figure 4.9: Comparison of Overhead due to Tasks Reallocation



(a) Temperature Difference (b) Average Number of Reallocated Tasks (c) Average Utilization of Reallocated Tasks

Figure 4.10: Comparison of Algorithms with Different Ambient Temperature. The x axis represents the range of ambient temperatures of processors. For each data point shown in this figure, the ambient temperatures of processors are in the range $[(45 - x/2)°C, (45 + x/2)°C]$.

At the single-node system level, a predictive dynamic thermal management method [82] has been proposed to regulate the temperature generated by multimedia applications. At the distributed system level, Weatherman [68] adopts a *data-driven* method to derive the heat distribution in data centers and then use this distribution to adjust workload in the data centers.

The work most closed related to this paper is Power Balancing [64]. It used an event counter to estimate the power of tasks on at run time and then performs thermal balancing through task migration in a multi-processor system. However, this work performs thermal balancing based on power estimation instead of direct temperature measurement. Thus different applications need different parameters that need to be calibrated. Furthermore, their algorithms are not analytically designed based on a control-theoretic approach, which is a key contribution of this work.

4.6 Summary

This paper proposes a control-theoretic approach to thermal balancing in server clusters. Our work has the following key contributions: (1) a formulation of thermal balancing as an optimal control problem; (2) a difference equation model that characterizes the thermal

dynamics for thermal balancing in server clusters; (3) two thermal balancing algorithms analytically designed based on optimal control theory; and (4) control analysis that establishes the stability and robustness of the control algorithms under system uncertainties in system power consumption. Simulation results demonstrate the capability of our control-theoretic approach to achieve thermal balancing under a wide range of uncertainties in terms of power consumption, ambient temperature, and thermal fault. By employing both temperature and CPU utilization feedbacks in its optimal control design, the CTB-UT algorithm provides a particularly attractive solution for server clusters, as it introduces low control cost for task reallocations and converges quickly to balanced temperatures.

Chapter 5

CloudPowerCap: Integrating Power Budget and Resource Management across a Virtualized Server Cluster

5.1 Introduction

In many datacenters, server racks are as much as 40 percent underutilized [30]. Rack slots are intentionally left empty to keep the sum of the servers' nameplate power below the power provisioned to the rack. And the servers placed in the rack cannot make full use of the rack's provisioned power. The root cause of this rack underutilization is that a server's peak power consumption is in practice often 40 percent lower than its nameplate power [22].

To address rack underutilization, server vendors have introduced support for per-host power caps, which provide a hardware or firmware-enforced limit on the amount of power that the server can draw [19, 39, 43]. These caps work by changing processor p-states or by using processor clock throttling, which is effective since the processor is the largest consumer of power in a server and its activity is highly correlated with the server's dynamic power consumption [22, 39]. Using per-host power caps, data center operators can set the caps on the servers in the rack to ensure that the sum of those caps does not exceed the rack's provisioned power. While this approach improves rack utilization, it burdens the operator with managing the rack power budget across the hosts. In addition, it does not lend itself to flexible allocation of power to handle workload spikes or to respond to the addition or removal of a rack's powered-on server capacity.

Many datacenters use their racked servers to run virtual machines (VMs). Several research projects have investigated power cap management for virtualized infrastructure [18, 53, 72, 73, 78, 98]. While this prior work has considered aspects of VM Quality-of-Service (QoS) in allocating the power budget, it has not explored a holistic power cap management framework to coordinate with a comprehensive production-quality resource management system for virtualized infrastructure. Such systems provide admission-controlled resource reservations, resource entitlements based on fair-share scheduling, load-balancing to maintain normalized host resource headroom for demand bursts, and respect for constraints to handle system heterogeneity and the user's business rules [34].

The operation of virtualized infrastructure resource management can be compromised if power cap budget management is not tightly coordinated with it. 1) Host power cap changes may cause the violation of VMs' resource reservations, impacting end-users' Service-Level Agreements (SLAs). 2) Host power cap changes may interfere with the delivery of VMs' resource entitlements, impacting resource fairness among VMs. 3) Host power cap changes may lead to imbalanced resource headroom across hosts, impacting peak performance and robustness in accommodating VM demand bursts. 4) Power cap settings can may limit the ability of the infrastructure to respect constraints, impacting infrastructure usability. 5) Resource management systems may support power proportionality via powering hosts off and on along with changing the level of VM consolidation. A discrepancy between the operation of power budget management and power proportionality may lead to the power budget being inefficiently allocated to hosts, impacting the amount of powered-on computing capacity available for a given power budget.

In this paper we present *CloudPowerCap*, a holistic and adaptive solution for power budget management in a virtualized environment. CloudPowerCap manages the power budget for a cluster of virtualized servers, dynamically resetting the per-host power caps for hosts in the cluster. The key of CloudPowerCap is to treat and manage the power cap in close *co-ordination* with resource management system. CloudPowerCap maps each host's power cap into resources capacity, by which CloudPowerCap can interoperate with a sophisticated resource management system of cloud datacenters, allowing it to manage power caps through the VM resource controls supported by resource management systems. CloudPowerCap

provides global fairness on dynamical power caps distribution with robustness for unpredictable workload variation, preventing hosts from gaining unfair entitlement of power caps and enhancing the system's capability to enforce VM placement constraints.

To the best of our knowledge, CloudPowerCap as proposed in this paper is the first holistic framework to provide dynamic power budget management in coordination with a cloud resource management system. The contributions of this paper are in four areas:

- We introduced the idea of converting a host's power cap to its CPU capacity, which is in turn managed as a *first class* computing resource by the cloud resource management system. This facilitates interoperability between power budget and resource management systems.
- We developed a mechanism to reallocate host power caps to satisfy constraints, including resource reservations and business rules.
- We designed power cap balancing among servers to provide fairness in terms of robustness to accomodate demand fluctuation. Power cap balancing can reduce or eliminate the need for moving VMs for load balancing, reducing the associated VM migration overhead.
- We designed power cap redistribution among servers to handle server power state changes caused by dynamic power management. Power cap redistribution reallocates the power budget freed up by powered-off hosts, while reclaiming budget to power-on those hosts when needed.

The rest of the chapter is organized as follows. Section 5.2 motivates the problem CloudPowerCap is addressing. Section 5.3 presents an overview of the CloudPowerCap design. Section 5.4 describes an implementation of CloudPowerCap. Section 5.5 shows experimental results for CloudPowerCap. Section 5.6 discusses related work. Section 5.7 provides summary of this chapter.

5.2 Motivation

In this section, we motivate the problem CloudPowerCap is intended to solve. We first discuss the trade-offs in managing a rack power budget. We then provide several examples of the value of combining dynamic rack power budget management with a cloud resource management system.

5.2.1 Managing a Rack Power Budget

To illustrate the problem of managing a rack power budget, we consider the case of a rack with a budget of 8 KWatt, to be populated by a set of servers. The server has 34.8 GHz CPU capacity comprising 12 CPUs, each running at 2.9 GHz and other parameters shown in Table 5.1. We note that server power consumption $P_{consumed}$ is commonly estimated by

CPU(GHz)	Memory(GB)	Nameplate(W)	Peak(W)	Idle(W)
34.8	96	400	320	160

Table 5.1: The configuration of the server in the rack.

a linear function of CPU utilization U and idle P_{idle} and peak P_{peak} power consumption of the host [22, 66] as

$$P_{consumed} = P_{idle} + (P_{peak} - P_{idle})U. \tag{5.1}$$

For a host power cap P_{cap} below its peak power, Equation (5.1) can be used to solve the CPU capacity C_{cap} reached at the power cap, i.e., the host's effective CPU capacity limit which we refer to as the *power-capped capacity*, given the peak CPU capacity C_{peak} corresponding to the peak power:

$$C_{capped} = C_{peak}(P_{cap} - P_{idle})/(P_{peak} - P_{idle}). \tag{5.2}$$

Given the above equation and the servers in Table 5.1, the rack's 8 KWatt power budget can accomodate various deployments including those shown in Table 5.2. Based on nameplate power, only 20 servers can be placed in the rack. Instead setting each server's power cap to its peak attainable power draw allows 25 percent more servers to be placed in the rack. This choice maximizes the amount of CPU capacity available for the rack power budget, since it best amortizes the overhead of the servers' powered-on idle power consumption. However, if

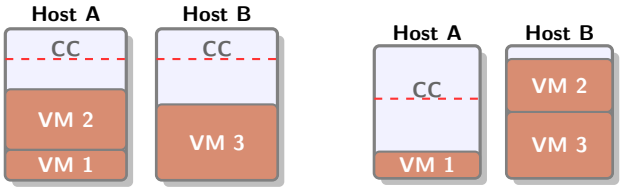
memory may sometimes become the more constrained resource, the memory made available by placing additional servers in the rack may be critical. Setting each server's power cap to 250 Watts allows 32 hosts to be placed in the rack, significantly increasing the memory available for the given power budget. By dynamically managing the host power cap values, CloudPowerCap allows trade-offs between CPU and memory capacity to be made at runtime according to the VMs' needs.

Power Cap(W)	**Count**	**CPU**		**Memory**	
		Capa(GHz)	**Ratio**	**Size(GB)**	**Ratio**
400	20	696	1.00	1920	1.00
320	25	870	1.25	2400	1.25
285	28	761	1.09	2688	1.40
250	32	626	0.90	3072	1.60

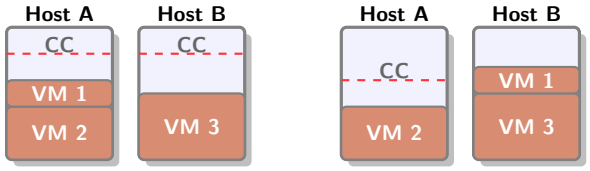
Table 5.2: Server deployments in a rack with 8 KWatt power budget with different power caps

5.2.2 Powercap Distribution Examples

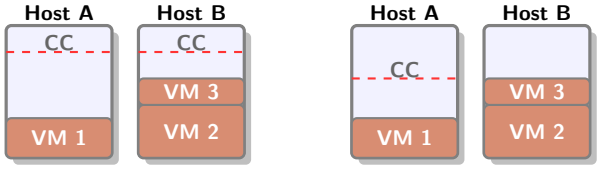
We use several scenarios to illustrate how CloudPowerCap can redistribute host power caps to support cloud resource management, including enabling VM migration to correct constraint violations, providing spare resource headroom for robustness in handling bursts, and avoiding migrations during load balancing. In these scenarios, we assume a simple example of a cluster with two hosts. Each host has an uncapped capacity of 2x3GHz (two CPUs, each with a 3GHz capacity) with a corresponding peak power consumption of 600W (values chosen for ease of presentation).



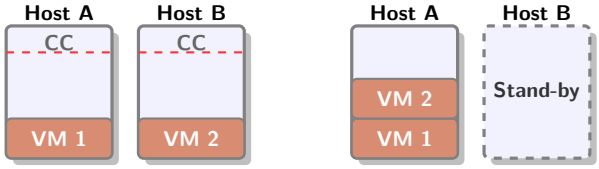
(a) Power cap redistribution enables VMs movement



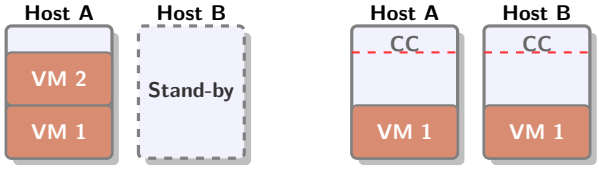
(b) Power cap redistribution improves robustness.



(c) Power cap redistribution reduces overhead of load balancing



(d) Power cap redistribution improves robustness after powering off a host



(e) Power cap redistribution balances robustness after powering on a host

Figure 5.1: Power cap distribution scenarios. Left-hand figures correspond to hosts status before distribution; right-hand figures show hosts status after. Power-capped capacity is not shown when the power cap of the host equals its peak power. (CC: Power-capped capacity)

Enforcing constraints: The host power caps should be redistributed when VMs are placed initially or relocated, to respect constraints and to allow resolution of constraint violations. For example, a cloud resource management system would move VM(s) from a host violating affinity constraints to a target host with sufficient capacity. However, in the case of static power cap management, this VM movement may not be feasible because of a mismatch between the VM reservations and the host capacity. As shown in Figure 5.1(a), host A and B have the same power cap of 480 W, which corresponds to a power-capped capacity of 4.8 GHz. Host A runs two VMs, VM 1 with reservation 2.4 GHz and VM 2 with reservation 1.2 GHz. And host B runs only one 3 GHz reservation VM. When VM 1 needs to be colocated with VM 3 due to a new VM-VM affinity rule between the two VMs, no target host in the cluster has sufficient power-capped capacity to respect their combined reservations. However, if CloudPowerCap redistributes the power cap of host A and B as 3.6 GHz and 6 GHz respectively, then VM 1 can successfully be moved by the cloud resource management system to host B to resolve the rule violation in the cluster. Note that host A's capacity cannot be reduced below 3.6 GHz until VM 1's migration to host B is complete or else the reservations on host A would be violated.

Enhancing robustness to demand bursts: Even when VM moves do not require changes in the host power caps, redistributing the power caps can still benefit the robustness of the hosts to handling VM demand bursts. For example, as shown in Figure 5.1(b), suppose as in the previous example that VM 1 needs to move from host A to host B because of a rule. In this case, a cloud resource management system can move VM 1 to host B while respecting the VMs' reservations. However, after the migration of VM 1, the *headroom* between the power capped capacity and VMs' reservations is only 0.6 GHz, compared with 2.4 GHz on host A. Hence, host B can only accommodate as high as a 15% workload burst without hitting the power cap while host A can accommodate 100%, that is, host B is more likely to introduce a performance bottleneck than host A. To handle this imbalance of robustness between two hosts, CloudPowerCap can redistribute the power caps of host A and B as 3.6 GHz and 6 GHz respectively. Now both hosts have essentially the same robustness in term of *headroom* to accommodate workload bursts.

Balancing load without VM migration: Before load balancing, power caps should be redistributed to reduce the need for VM migrations. Load balancing of the resources to which the VMs on a host are entitled is a core component of cloud resource managment

since it can avoid performance bottlenecks and improve system-wide throughput. However, some recommendations to migrate VMs for load balancing among hosts are unnecessary, given that power caps can be redistributed to *balance* workload, as shown in Fig 5.1(c). In this example, the VM on Host A has an entitlement of 1.8 GHz while the VMs on host B have a total entitlement of 3.6 GHz. The difference in entitlements between host A and B are high enough to trigger load balancing, in which VM 3 is moved from host B to host A. After load balancing, host A and B have entitlements of 3 GHz and 2.4 GHz respectively, that is, the workloads of both hosts are more balanced. However, VM migration has an overhead cost and latency related to copying the VM's CPU context and in-memory state between the hosts involved [84], whereas changing a host power cap involves issuing a simple baseboard management system command which completes in less than one millisecond [39]. CloudPowerCap can perform the cheaper action of redistributing the power caps of hosts A and B, increasing host B's power capped capacity to 6 GHz after decreasing host A's power capped capacity to 3.6 GHz, which also results in more balanced entitlements for host A and B. The redistribution of power cap before load balancing, called *powercap balancing*, can reduce or eliminate the overhead associated with VM migration for load balancing, while introducing no compromise in the ability of the hosts involved to satisfy the VMs' resource entitlements. We note that the goal of load balancing is not *absolute* balance of workload among hosts, which may not be possible or even worthwhile given VM demand variability, but rather reducing the imbalance of hosts' entitlements below a predefined threshold [34].

Adapting to host power on/off: Power caps should be redistributed when cloud resource management powers on/off host(s) to improve cluster efficiency. A cloud resource management system detects when there is ongoing under-utilization of cluster host resources leading to *power-inefficiency* due to the high host idle power consumption, and it consolidates workloads onto fewer hosts and powers the excess hosts off. In the example shown in Figure 5.1(d), host B can be powered off after VM 2 is migrated to host A. However, after host B is powered-off, it does not consume power and hence not need its power cap. And the utilization of host A is increased due to migrated VM 2, which impacts the capacity *headroom* of host A. Power cap redistribution after powering off host B can increase the power cap of host A to 6 GHz, allowing the *headroom* of host A to increase to 3 GHz and hence increase system robustness and reduce the likelihood of resource throttling.

On the other hand, if there are *overloaded* hosts in the cluster, cloud resource management powers on stand-by hosts to avoid performance bottleneck as seen in Figure 5.1(e). Due to dynamic power cap management, active hosts can fully utilize the cluster power cap for robustness. So a host being powered-on may not have enough power cap to run VMs migrated to it with suitable robustness. CloudPowerCap can handle this issue by redistributing the power cap among the active hosts and the host exiting standby appropriately. For example, as shown in Figure 5.1(e), host B is powered on because of the high utilization of host A, and can only acquire 3.6 GHz power-capped capacity due to the limit of the cluster power budget. If VM 2 migrates to the host B to offload the heavy usage of host A, the *headroom* of the host B will only be 1.2 GHz, contrasting to the *headroom* of host A, 3.6 GHz. However, after power cap redistribution, the power caps of host A and B can be assigned to 4.8 GHz respectively, balancing the robustness of both hosts.

5.3 CloudPowerCap Design

In this section, we first introduces how CloudPowerCap maps a power cap to CPU capacity. Then the principles of CloudPowerCap design are presented. Finally we briefly describe overview structure and the components in CloudPowerCap .

5.3.1 CloudPowerCap Power Model

The power model adopted by CloudPowerCap maps the power cap of the host to the CPU capacity of the host, which is in turn managed by a resource management system directly. To efficiently and conservatively calculate the mapping between host capacity and power consumption, CloudPowerCap employs a linear power model between CPU utilization and host power consumption as shown in Figure 5.2. The power P_{idle} represents the power consumption of the host when the CPU is idle. P_{idle} intentionally includes the power consumption of the non-CPU components, such as memory and spinning disk, since their power draw does not vary significantly with utilization. System platforms often report P_{idle} or it can be collected via a one-time calibration step. The power P_{peak} represents the power consumption of the host when the CPU is 100% utilized at its peak GHz rating. Again, this value may

be collected during a one-time calibration step (if it can be attained given the host's power cap) or it may be estimated based on power consumption measurements taken at operating points in the attainable CPU GHz range. Using the line defined by these two values and the model that the power consumption of the host tracks CPU utilization, we estimate the CPU capacity associated with a host power cap value.

We note that since P_{peak} is measured when CPU is running at full speed and given power saving features such as P-states (DVFS), the model shown in Figure 5.2 can be a under-estimation of the CPU capacity associated with a particular level of power consumption. A more accurate CPU power model [22] could be integrated into CloudPowerCap to reduce its conservativeness.

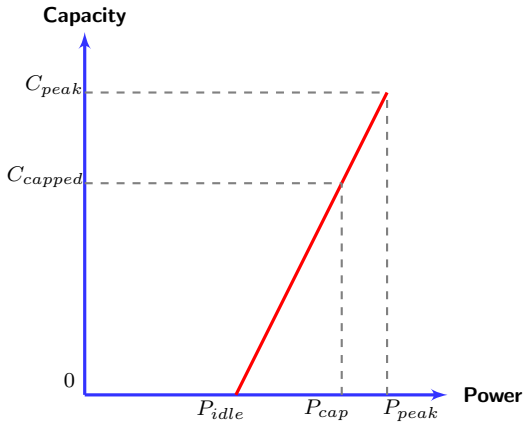


Figure 5.2: Mapping a power cap (P_{cap}) to capped capacity (C_{capped}). P_{idle} and P_{peak} are the idle and peak power of a host respectively. C_{peak} and C_{capped} are the uncapped and capped raw capacity respectively.

When computing power-capped capacity of a host based on the power model shown in Fig. 5.2, it is important to ensure that the capacity reserved by the hypervisor on the host is fully respected. Hence, the power-capped capacity $C_{mcapped}$ managed by the resource management system, i.e., managed capacity, is computed as:

$$C_{mcapped} = C_{capped} - C_H, \tag{5.3}$$

where the power-capped raw capacity C_{capped} is computed using Equation (5.1) and C_H is the capacity reserved by the hypervisor. In later sections we always refer power-capped capacity as the manageable power-capped capacity $C_{mcapped}$.

5.3.2 CloudPowerCap Design Priciples

The primary goal of CloudPowerCap is to provide dynamic power caps management of virtualized servers with an *existing* resource management system, delivering efficient resources usage, performance isolation and respecting the cluster power budget. The *existing* resource management systems in virtualized environment is dedicated to solve the problem how to assign and dynamically scheduling VMs over hosts with *fixed* capacity. The concept of power caps adds another dimension to the solution of this problem, i.e., changing capacity of hosts rather than allocating VMs over hosts to satisfied performance and other conditions of cloud infrastructure. *Extra* capacity induced by power cpas can be used as to accommodate transient workload bursts without involving expensive VM migration. Although an overhaul of existing resource management system implemented on hosts with *fixed* capacity to understand and leverage poewr caps may be preferable to provide a holistic approach to manage resources with power caps, this approach is, if not impossible, at least significantly difficult to apply due to the cost of redesign and implementation of a production system. Hence we adopt an approach with a more practical and efficient manner. In CloudPowerCap we separate the problem of resource management with power cap into two parts : 1) resources management on hosts with *fixed* capacity, handled by updated DRS with awareness of power caps; 2) power cap management to redistribute power caps over hosts, performed by CloudPowerCap in coordination with DRS.

5.3.3 CloudPowerCap Overview

The resource management systems of virtualized server clusters are designed to achieve service performance objectives by properly allocating resources to virtual machines under multi-tenancy environment. The major functions of resource management systems are the following:

VMs Placement: VMs placement involves initial placement of VMs and relocation of VMs for constraints correction to respect user defined or business rules. During initial placement, the proper physical hosts are selected to launch newly generated VMs. The user defined and business rules restrict VMs locations on physical hosts.

Load Balancing: Load balancing continuously responds to workload imbalance by migrating VMs between hosts to alleviate potential performance bottleneck.

Power Management: To improve power efficiency, the workload distributed over hosts may be consolidated on a subset hosts while remaining hosts are turned off to save energy. On the other hand, hosts may be turned on at run time to handle workload bursts.

Sanity Check: During invocation of resource management, the status of cluster's resources may be changed due to hosts failure or maintenance. To provide a *clean* ground for other functions, the sanity check function checks status of hosts and updates resources associated with hosts.

Some commercial cloud resources management systems, for examples, VMware Distributed Resource Scheduler (DRS) and Microsoft Virtual Machine Manager (VMM), implemented all above functions. Although most open-source resource management systems currently only equip with simple mechanism for initial VMs placement, extension including load balancing and power management are proposed in previous research [6, 101].

Since the aim of CloudPowerCap is to enforce the cluster power budget while dynamically managing power caps of hosts by closely coordinating with a cloud resource management system, the structure of CloudPowerCap consists of four components, as shown in Figure 5.3, corresponding to the four major functions of the resources management systems.

Powercap Allocation: During the CloudPowerCap power cap allocation phase, potential resource management constraint correction moves may require redistribution of host power caps. Note that because CloudPowerCap can redistribute the host power caps, the cloud resource management system can correct more constraint violations than would be possible with statically-set host power caps.

Powercap Balancing: During the load-balance analysis stage, if the resource management system detects load imbalance over the user-set threshold, Powercap balancing first tries to reduce the imbalance. Powercap balancing can lower load imbalance by redistributing power caps without needing to migrate VMs between hosts. This is valuable because VM live migration engenders CPU and memory overhead on both the source and target hosts to send the VM's virtual device state, to update its external device connections, to copy its

memory one or more times to the target host while tracing the memory to detect any writes requiring recopy, and to make the final switchover [90]. While the migration cost may be transparent to the VMs if there is sufficient host headroom, reducing or avoiding the cost when possible increases efficiency. Powercap Balancing may not be able to fully address imbalance due to inherent physical host power consumption limits. If power cap balancing is not able to reduce the imbalance below the specified imbalance threshold, the resource management load balancing function of the resource management can address the remaining imbalance by VM migration.

During CloudPowerCap initialization, for each host, the mapping between its current power cap and its effective capacity is established by the mechanisms described in Section 5.3.1. For a powered-on host, the power cap value should be in the range between the host's idle and peak power.

Powercap Redistribution: If the cloud resource management system considers powering on a host to match a change in workload demands or other requirements, CloudPowerCap performs a two-pass power cap redistribution. First it attempts to re-allocate sufficient power cap for that host to power-on. If that is successful and if the system selects the host in question after its what-if power-on evaluation, then CloudPowerCap redistributes the cluster power cap across the updated hosts, to address any unfairness in the resulting power cap distribution. Similarly, if the system considers powering off a host, its power cap can be redistributed fairly to the remaining hosts after the completion of the host power-off operation.

Powercap Check: Corresponding to *Sanity Check* in resource management, during the CloudPowerCap check phase, CloudPowerCap distributes unallocated cluster power due to hosts in failure or maintenance to hosts whose power caps are less than the peak power.

We note that CloudPowerCap can be used with various cloud resource management systems. As discussed in the next section, we have implemented and integrated CloudPowerCap with VMware Distributed Resource Scheduler (DRS) [88] and VMware Distributed Power Management (DPM) [89], but the general strategy of CloudPowerCap can complement other distributed resource management systems for virtualization environments. For example, the OpenStack computing nodes scheduler [74] assigns VMs to hosts considering constraints during its filtering step and memory footprint during its weight and cost step. Although

OpenStack does not support CPU reservations or the notion of estimating and satisfying CPU demand, it does track a vCPU overcommitment ratio, defined to be the number of vCPUs per pCPU. CloudPowerCap could interoperate positively with OpenStack by redistributing the rack power budget across a set of hosts in accordance with the degree of vCPU overcommitment. This is particularly beneficial for OpenStack environments, which do not use automatic live migration for load-balancing.

5.4 CloudPowerCap Implementation

We implemented CloudPowerCap to work with the DRS cloud resource management system. Figure 5.3 shows the high-level structure of CloudPowerCap working with DRS. In this section, we first present an overview of DRS and then detail the design of each CloudPowerCap component and its interaction with its corresponding DRS component.

5.4.1 DRS Overview

VMware DRS performs resource management for a cluster of ESX hypervisor hosts. It supports a rich set of controls for efficient multi-resource management to provide differentiated QoS to VMs. The basic resource controls in DRS (listed below) allow users to express resource allocation in terms of *guaranteed service-rate* and/or *relative importance* assuming a mapping between service level and resources.

Reservation: A reservation specifies the minimum amount of CPU or memory resources guaranteed, even if the cluster is over-committed.

Limit: A limit specifies the upper bound of CPU or memory resources allocated, even if the cluster is under-committed.

Shares: Shares express *relative importance* and represent weights of resouce allocation used if there is resource contention.

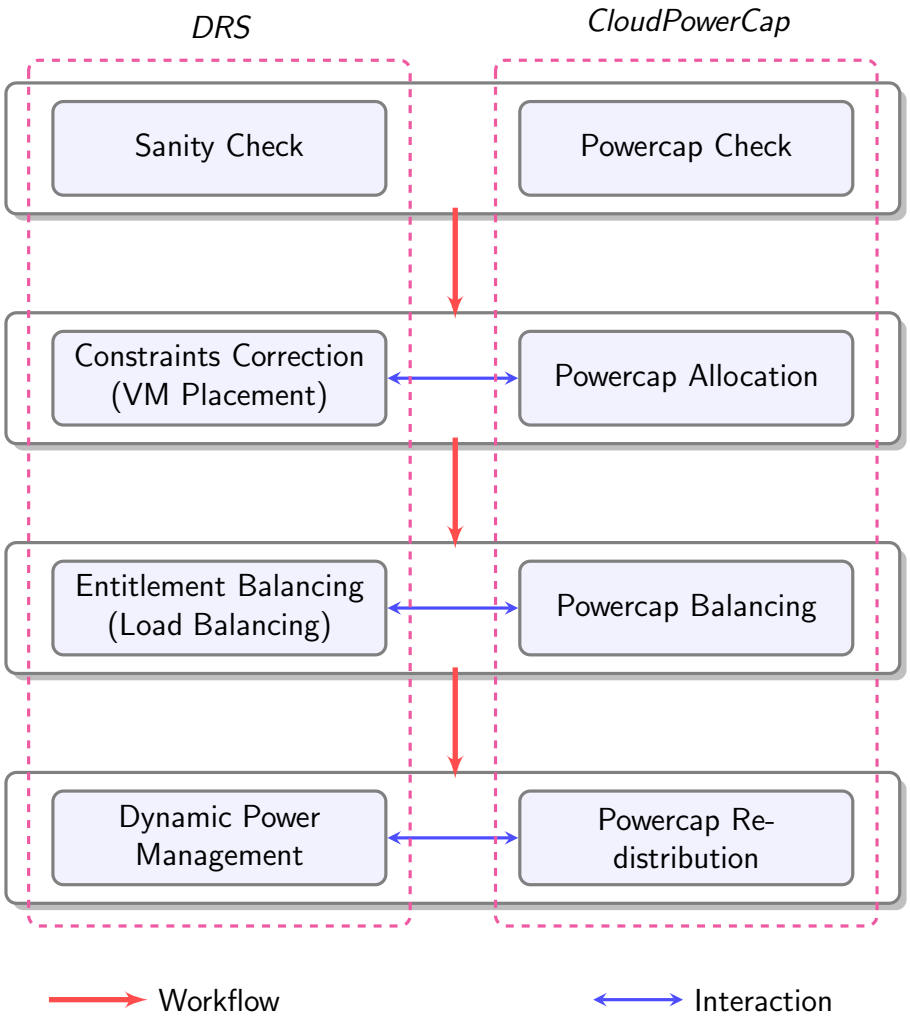


Figure 5.3: Structure of CloudPowerCap working with DRS. *Italic* texts indicate corresponding components in general resource management systems.

The DRS resource controls can be set on an individual VM or on a *resource pool*, an aggregation of resources allowing performance isolation and resource sharing between a set of VMs. Resource entitlements are computed according to these resource controls along with an estimate of VM CPU and memory resource *demand*, a key metric intended to capture the amount of CPU and memory the VM would use to satisfy its workload if there were no contention. The demand that DRS works with is smoothed over an extended period given the relatively coarse granularity at which DRS runs.

By default, DRS is invoked every five minutes. It evaluates the state of the cluster and considers recommendations to improve that state by executing those recommendations in a what-if mode on an internal representation of the cluster. At the end of each invocation, DRS issues zero or more recommendations for execution on the actual cluster.

At the beginning of each DRS invocation, DRS executes resource sanity check code to detect any inconsistencies between admitted resource reservations and current cluster capacity.

Next, DRS corrects any cluster constraint violations by migrating VMs between hosts. Examples of such corrections include evacuating hosts that the user has requested to enter maintenance or standby mode and ensuring VMs respect user-defined affinity and anti-affinity rules. Both sanity check and constraint correction aim to create a *constraint compliant snapshot* of the cluster for further DRS processing. We note that choosing a host on which to power-on a VM is treated as a special-case constraint violation.

DRS next performs entitlement balancing. DRS employs *normalized entitlement* as the load metric of each host. Denoted by N_h, *normalized entitlement* is defined as the sum of the per-VM entitlements E_i for each VM running on the host h, divided by the capacity of the host, C_h, i.e., $N_h = \frac{\sum E_i}{C_h}$. DRS's entitlement balancing algorithm uses a greedy hill-climbing technique with the aim of minimizing the overall cluster load imbalance (i.e., the standard deviation of the hosts' normalized entitlements). DRS chooses as each successive move the one that reduces load imbalance most, subject to a risk-cost-benefit filter which considers workload stability risk and VM migration cost versus the increased balance benefit. The move-selection step repeats until either the load imbalance is below a user-set threshold, no beneficial moves remain, or the number of moves generated in the current pass hits a configurable limit based on an estimate of how many can be executed in five minutes.

DRS then optionally runs DPM, which opportunistically saves power by dynamically right-sizing cluster capacity to match recent workload demand, while respecting the cluster constraints and resource controls. DPM recommends evacuating and powering off host(s) if the cluster contains sufficient spare resources, and powering on host(s) if either resource demand increases appropriately or more resources are needed to meet cluster constraints.

5.4.2 Powercap Check

Powercap Check generates recommendations to distribute any unallocated cluster power budget among powered-on hosts whose power caps are less than their peak capacity. The unallocated cluster power budget may change in several cases : 1) immediately after the initial setup of the host power caps and the cluster power budget; 2) hosts are placed in standby or removed from the rack; 3) cluster power budget is increased by clients. Each powered-on host below peak capacity is given an increase in power cap proportional to the ratio of the unallocated cluster power budget to the amount needed by all powered-on hosts until it reaches peak capacity. The purpose of the Powercap Check step is to use excess cluster power budget to increase the effective capacity of the powered-on hosts.

5.4.3 Powercap Allocation

After the unallocated power budget has been distributed, Powercap Allocation redistributes power caps if needed to allow DRS to correct constraint violations.

DRS hard constraints include evacuating VMs from hosts that the user has requested to enter maintenance or standby mode, complying with mandatory VM-to-VM or VM-to-Host affinity and anti-affinity rules, and ensuring VM CPU and memory reservations are met. DRS's ability to correct constraint violations is impacted by host power caps , which can limit the available capacity on target hosts. However, as shown in Fig 5.1(a), by increasing the host power cap, the DRS algorithm can be more effective in correcting constraint violations. Hence to aid DRS constraint correction, Powercap Allocation supports redistributing the cluster's unreserved power budget , i.e., the amount of power not needed to support running VMs' CPU and memory reservations. The unreserved power budget represents the maximum

amount of power cap that can be redistributed to correct violations; insufficient unreserved power budget prevents the correction of constraint violations.

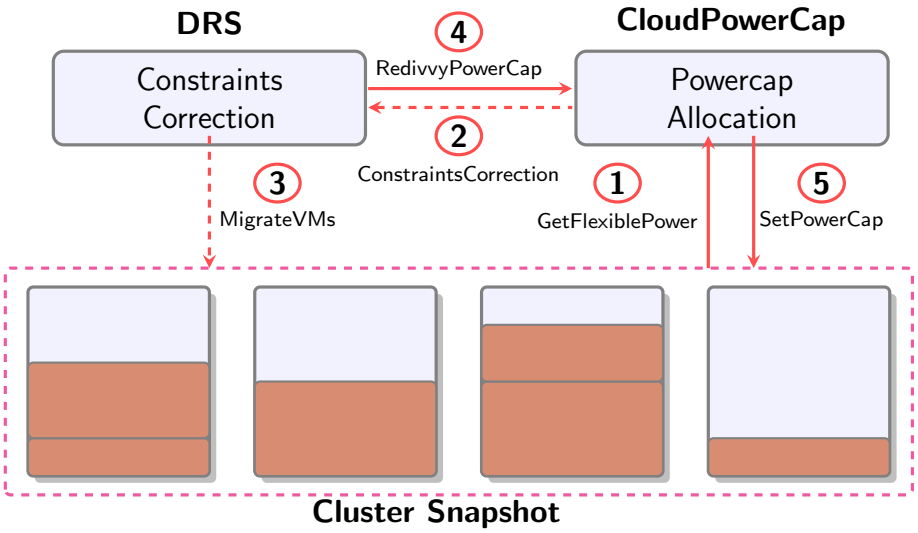


Figure 5.4: Coordination between CloudPowerCap and DRS to correct constraints. Solid arrow indicates invocations of CloudPowerCap functions while dashed arrow indicates invocations of DRS functions.

CloudPowerCap and DRS works in coordination, as shown in Figure 5.4, to enhance the system's capability to correct constraints violations.

1) Powercap Allocation first calls *GetFlexiblePower* to get *flexiblePower*, which is a special clone of the current cluster snapshot in which each host's host power cap is set to its reserved power cap, i.e., the minimum power cap needed to support the capacity corresponding to the reservations of the VMs currently running on that host.

2) The *flexiblePower* is used as a parameter to call *ConstraintsCorrection* function in DRS, which recommends VM migrations to enforce constraints and update hosts' reserved power caps for the new VM placements after the recommended migrations. Then DRS generates an action plan for migrating VMs.

3) After performing *ConstraintsCorrection*, DRS generates VM migration actions to correct constraints. Note when applying VMs migration actions on hosts in the cluster, precedence requisites are followed between these actions and power cap setting actions.

4) If some constraints are corrected by DRS, the power caps of source and target hosts need reallocated to ensure fairness. For this case, *RedivvyPowerCap* of CloudPowerCap is called to redistribute the power cap.

5) Finally Powercap Allocation generates actions to set power cap of hosts in the cluster according to the results of *RedivvyPowerCap*.

The key function in Powercap Allocation is Powercap Redivvy, in which the unreserved power budget is redistributed after the operations for constraint violation correction. An algorithm used for Powercap Redivvy is presented in Algorithm 2. The input to this step are S, the current snapshots of the cluster and updated snapshot F, in which the cluster power budget has been distributed according to *proportional resource sharing* [91] to maintain fairness of unreserved power budget distribution across hosts. The actions to change host power cap on hosts are also generated if the hosts need more power cap than those in S or less power cap without violating VM reservation. Note these sets of power cap changes are made appropriately dependent of the actions generated by DRS to correct the constraint violations.

Algorithm 2 Powercap Allocation

S, F: cluster snapshots before and after constraints correction;
$C_{i,S}, C_{i,F}$ power cap of the host h_i in S and F;

```
1: function RedivvyPowerCap(S, F)
2:     C_needed ← 0, C_excess ← 0
3:     for each host h_i in the cluster do
4:         if C_{i,F} > C_{i,S} then
5:             SetPowerCap(h_i, C_{i,F})
6:             C_needed ← C_needed + (C_{i,F} − C_{i,S})
7:         else
8:             C_excess ← C_excess + (C_{i,S} − C_{i,F})
9:         end if
10:    end for
11:    if C_needed > 0 then
12:        r ← C_needed/C_excess
13:        for each host h_i in the cluster do
14:            if C_{i,F} ≤ C_{i,S} then
15:                C_{i,F} ← C_{i,F} + r(C_{i,S} − C_{i,F})        ▷ Proportional sharing
16:                SetPowerCap(h_i, C_{i,F})
17:            end if
18:        end for
19:    end if
20: end function
```

5.4.4 Powercap Balancing

Load balancing is critical for systems managing distributed resources, to maintain fairness and to improve the responsiveness to bursts in resource demand and achieved by migrating workload between hosts. For resource management system without concept of power caps, like DRS, load balancing achieves both of these goals by reducing imbalance via migrating VMs between hosts. However, with dynamic power cap management, CloudPowerCap can alleviate imbalance by increasing the power caps of heavy loaded hosts while reducing the power caps of lightly loaded hosts rather than migrating VMs migration between those hosts as shown in Figure 5.1(c). Considering almost negligible overhead of power cap reconfiguration comparing to VMs migration, Powercap Balancing is preferred to DRS entitlement balancing once workload of the cluster is imbalanced. Nevertheless, because of limitation of power cap adjustment, Powercap Balancing may not eliminate all imbalance of the cluster. But the amount of VMs migration involved in subsequent load balancing can be reduced significantly.

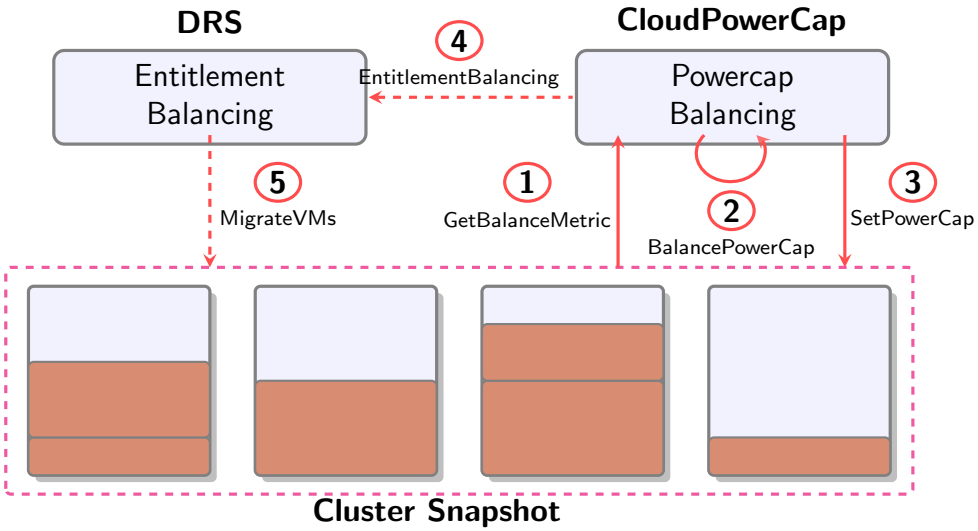


Figure 5.5: Work flow of Powercap Balancing and its interaction with DRS load balancing. Solid arrow indicates to invocations of CloudPowerCap functions while dashed arrow indicates to invoke DRS functions.

The process of powercap balancing and its interaction with DRS load balancing are shown in Figure 5.5.

1) To acquire the status of entitlement imbalance of the cluster, Powercap Balancing first calculates balance index defined in DRS of the cluster (i.e., the standard deviation of the hosts normalized entitlements).

2) Then Powercap Balancing tries to reduce the entitlement imbalance among hosts by adjusting their power caps of hosts in accordance with their entitlement.

3) If Powercap Balancing is able to impact cluster imbalance, its host power cap redistribution actions are added to the list, with the host power cap reduction actions being prerequisites of the increase actions.

4) If Powercap Balancing has not fully balanced the entitlement among the hosts, DRS entitlement balancing is invoked on the results of Powercap Balancing to reduce entitlement imbalance further.

5) DRS generate actions to migrate VMs.

The sketch of the key function *BalancePowerCap* in Powercap Balancing is shown in Algorithm 3, which was developed along the lines of *progressive filling* to achieve max-min fairness [8]. The algorithm progressively increases the host power cap of the host(s) with highest normalized entitlement while progressively reducing the host power cap of the host(s) with lowest normalized entitlement . This process is repeated until either the DRS imbalance metric crosses the balance threshold or any of the host(s) with highest normalized entitlement reach their peak capacity and hence further reduction in overall imbalance is limited by those hosts.

5.4.5 Powercap Redistribution

Powercap Redistribution is used to in response to DPM dynamically powering on/off hosts. When CPU or memory utilization becomes high, DPM recommends powering on hosts and redistributing the VMs across the hosts to reduce per-host load. Before the host is powered on, Powercap Redistribution ensures that sufficient power cap is assigned to the power-on host. On the other hand, when both CPU and memory utilization are low for a sustained period, DPM may recommend consolidating VMs onto fewer hosts and powering off the remaining hosts to save energy. In this case Power Redistribution distributes the power cap of the powered-off hosts among the active hosts to increase their capacity.

Algorithm 3 Powercap Balancing

S, F: cluster snapshot before and after Powercap Balancing
h, l: hosts with highest and lowest normalized entitlement
$\hat{C}_i$: peak capacity of the host i
$\bar{C}_i$: capacity of the host i corresponding to average normalized entitlement of the cluster

1: **function** BALANCEPOWERCAP(S)
2: $F \leftarrow S$, pcBal $\leftarrow$ false
3: **while** Cluster is imbalanced **do**
4: Choose h and l from the cluster
5: $C_{needed} \leftarrow min(\hat{C}_h, \bar{C}_h) - C_h$
6: $C_{avail} \leftarrow C_l - \bar{C}_l$
7: **if** $C_{needed} = 0$ or $C_{avail} = 0$ **then**
8: break ▷ Then invoke DRS load balancing
9: **else**
10: pcBal $\leftarrow$ true
11: **end if**
12: Add C_{avail} to h and reduce C_{needed} from l
13: Recompute cluster balance metric on F
14: **end while**
15: **if** pcBal = true **then**
16: Set power cap of hosts according to F
17: **end if**
18: return F
19: **end function**

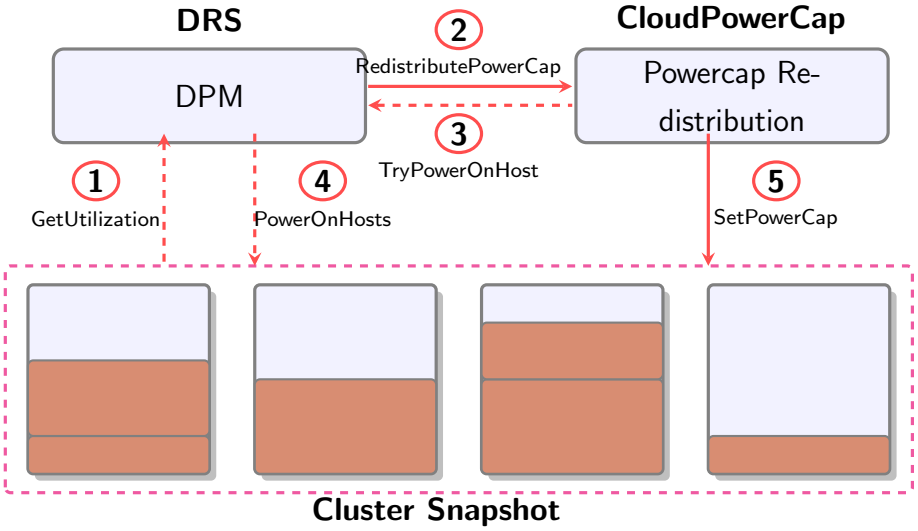


Figure 5.6: Coordination between CloudPowerCap and DRS and DPM in response to power on/off hosts. Solid arrow indicates to invoke CloudPowerCap functions while dashed arrow indicates to invoke DRS functions.

The coordination between Powercap Redistribution and DPM when DPM attempts to power on a host is depicted in Figure 5.6.

1) If there is sufficient unreserved cluster power budget to set the target host's power cap to peak, the host obtains its peak host power cap from the unreserved cluster power budget and no power cap redistribution is needed.

2) If the current unreserved cluster power budget is not sufficient, *RedistributePowerCap* is invoked to allow the power-on host acquiring more power caps from those hosts with low CPU utilization.

3) DPM decides whether to power on the host given its updated power cap after redistribution.

4) If the host is chosen for power-on, the normal DPM function is invoked to generate the action plan for powering on the host.

5) If DPM decides to recommend the candidate power-on, the host power cap changes are recommended as prerequisites to the host power-on.

The algorithm of redistributing power caps is straightforward. To acquire sufficient power caps to power on a host, the hosts with low utilization suppress their power caps under the constraint of not causing those hosts to enter the high utilization range that would trigger DPM to power on another host.

Algorithm 4 Powercap Redistribution for DPM power-on

S, F: cluster snapshots before and after Powercap Redistribution
p: standby host
C_u: capacity corresponding to unreserved cluster budget
$C_i, C_{i,on}$: current capacity and capacity corresponding to the threshold of power-on of the host i
$\hat{C}_i$: peak capacity of the host i

```
1: function RedistributePowerCap(S, h_sb)
2:     F ← S, C_needed ← Ĉ_p − C_p
3:     C_p ← C_p + C_u, C_u ← 0, C_needed ← C_needed − C_u
4:     for host i with utilization less than to power-on do
5:         C_u ← C_u + C_i − C_i,on
6:         if C_u ≥ C_needed then
7:             break
8:         end if
9:     end for
10:    if C_u − C_needed ≥ 0 then
11:        C_p ← C_p + C_needed, C_u ← C_u − C_needed
12:    else
13:        C_p ← C_p + C_u, C_u ← 0
14:    end if
15:    return F
16: end function
```

When a host is being considered for power-off, the portion of its host power cap currently above its utilization could be made available for redistribution to other powered-on hosts whose host power caps are below peak, to provide more target capacity for evacuating VMs.

5.4.6 Implementation Details

We implemented CloudPowerCap on top of VMware's production version of DRS. Like DRS, CloudPowerCap is written in C++. The entire implementation of CloudPowerCap comprises less than 500 lines of C++ code, which demonstrates the advantage of instantiating power

budget management as a separate module that coordinates with an existing resource manager through well defined interfaces.

As described previously in this section, DRS operates on a snapshot of the VM and host inventory it is managing. The main change we made for DRS to interface with CloudPowerCap was to enhace the DRS method for determining a host's CPU capacity to reflect the host's current power cap setting in the snapshot. Other small changes were made to support the CloudPowerCap functionality, including specifying the power budget, introducing a new action that DRS could issue for changing a host's power cap, and providing support for testability.

The implementation of power cap check was straightforward and only involved adding a new method called at the beginning of each DRS invocation to redistribute any unallocated power budget among the powered-on hosts by increasing their power caps in the snapshot. Powercap allocation entailed updating corresponding DRS methods to understand that a host's effective capacity available for constraint correction could be increased using the unreserved power budget, and adding a powercap redivvy step optionally run at the end of the constraint correction step. Powercap balancing, which leverages elements of the powercap redivvying code, involved creating a new method to be called before the DRS balancing method. Powercap redistribution changed DPM functions to consider whether to turn on/off hosts based not only on utilization but also on the available power budget.

5.5 Evaluation

In this section, we evaluate CloudPowerCap in the DRS simulator under three interesting scenarios. The first experiments evaluate CloudPowerCap 's capability to rebalance normalized entitlement among hosts while avoiding the overhead of VM migration. The second experiment shows CloudPowerCap reallocates the power budget of a powered-off host to allow hosts to handle demand bursts. The third experiment shows how CloudPowerCap supports CPU and memory capacity trade-offs to be made at runtime. This experiment includes a relatively large host inventory to show the capacity trade-offs at scale.

In these experiments, we compare CloudPowerCap against two baseline approaches of power cap management: *StaticHigh* and *Static*. Both approaches assign equal power cap to each host in the cluster at the beginning and maintain the power cap during the cluster running. *StaticHigh* sets power cap of the host as its peak power, maximizing throughput of CPU intensive applications. However for applications in which memory or storage become constrained resources, it can be beneficial to support more servers to provision more memory and storage. Hence in *Static*, the power cap of a host is intentionally assigned lower than the peak power of the host. Comparing *StaticHigh*, more servers are placed in *Static* to enhance the throughput of applications with memory or storage as constrained resources. However both approaches lack the ability of flexible allocation of power caps and can not respond to workload spikes and demand variation.

5.5.1 DRS Simulator

The DRS simulator is used in developing and testing all DRS algorithm features. It provides a realistic execution environment, while allowing much more flexibility and precision in specifying VM demand workloads and obtaining repeatable results than running on real hardware. The simulator is described in detail in Section 5.1.1 of [35]; we include an overview of its key features here.

The DRS simulator simulates a cluster of ESX hosts and VMs. It supports defining different host and VM profiles in order to experiment with different configurations. A host can be defined using parameters including number of physical cores, CPU capacity per core, total memory size, and power consumption at idle and peak. A VM can be defined in terms of number of configured virtual CPUs (vCPUs) and memory size. Each VM's workload can be described by an arbitrary function over time, with the simulator generating CPU and memory demand for that VM based on the specification.

Given the input characteristics of ESX hosts and the VMs' resource demands and specifications, the simulator mimics ESX CPU and memory schedulers, allocating resources to the VMs in a manner consistent with the behavior of ESX hosts in a real DRS cluster. The simulator supports all the resource controls supported by the real ESX hosts, including reservation, limit and shares for each VM along with the resource pools.

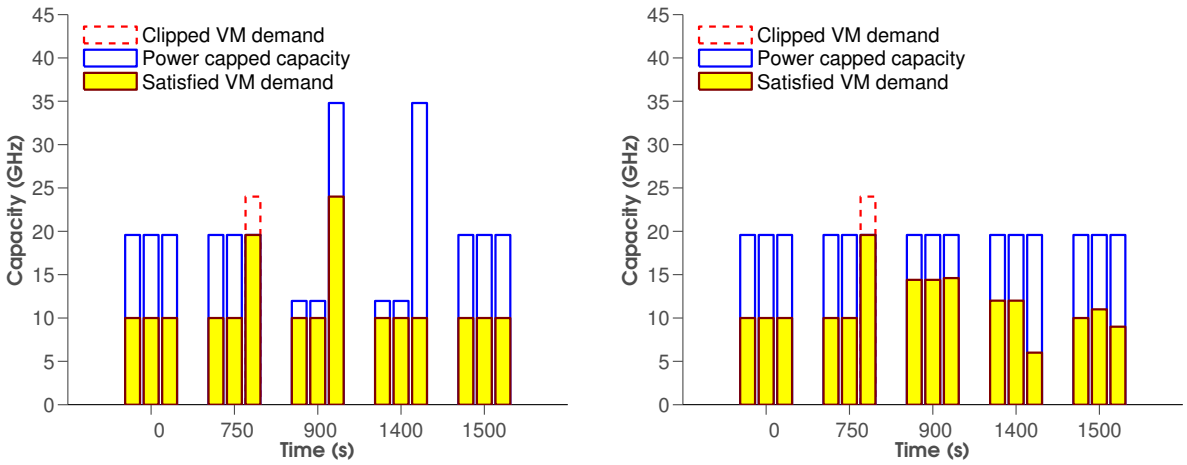
The simulator determines the allocation each VM receives, whenever there is any change in demand by any of the VMs on the host. The simulator supports vMotion of VMs, and models the cost of vMotion and its impact on the workload running in the VM, based on how vMotion works in a physical ESX host. The simulator also takes into account the resource settings for the resource pool trees on the host when resources are divvyed out, similar to how the real ESX host divvies out the host resources based on the host-level resource pool hierarchy. The simulator models the ESX hypervisor CPU and memory overheads.

The simulator also estimates power consumption of the ESX hosts based on the power model given in Equation (5.1) in Section 5.2.1. For this work, the simulator was updated to respect the CPU capacity impact associated with a host's power cap.

5.5.2 Headroom Rebalancing

CloudPowerCap can reassign power caps to balance headroom for bursts, providing a quick response to workload imbalance due to VM demand changes. Such reassignment of power caps can improve robustness of the cluster and reduce or avoid the overhead of VM migration for load balancing. To evaluate impact of CloudPowerCap on headroom balancing, we perform an experiment in which 30 VMs with 1vCPU and 8GB memory run 3 hosts with configuration shown in Table 5.1. Figures 5.7(a) and 5.7(b) plot the simulation results under CloudPowerCap and Static with a static power cap allocation of 250W per host, respectively. Initially, at time 0 seconds, the VMs are each executing similar workloads of 1 GHz CPU and 2 GB memory demand, and are evenly distributed across the hosts. At time 750 seconds, the VMs on one host spike to 2.4 GHz demand, thereby increasing the demand on that host above its power-capped capacity. When DRS is next invoked at time 900 seconds (running every 300 seconds by default), its goal is to rebalance the hosts' normalized entitlements. Under the static power cap, DRS migrates the VMs to balance the normalized entitlements. In contrast, CloudPowerCap reassigns the hosts' power caps to reduce the caps on the light-loaded hosts (to 215W) and increase them on the heavy-loaded host (to 320W). This addresses the host overutilization and imbalance without requiring vMotion latency and overhead, which is particularly important in this case, since the overhead further impacts the workloads running on the overutilized host. At time 1400, the 2.4 GHz VM demand spike ceases, and those VMs resume running at their original 1 GHz demand until the experiment

ends at time 2100. Again, CloudPowerCap avoids the need for migrations by reassigning the host power caps to their original values. In contrast, Static performs two entitlement balancing and migrates several VMS at time 900 and 1500.



(a) CloudPowerCap (b) Static

Figure 5.7: Headroom balancing on a group of 3 hosts. Hosts are gouped at each event time.

	CPU Payload Ratio	**vMotion**
CPC	0.99	0
Static	0.89	7
StaticHigh	1.00	0

Table 5.3: CloudPowerCap (CPC) rebalancing without migration overhead

Table 5.3 compares the CPU payload delivered to the VMs under CloudPowerCap , Static using 250W static host power caps, as well as StaticHigh using the power caps equivalent to the peak capacity of the host. For Static, the vMotion CPU overhead has a significant overall impact on the CPU payload delivered to the VMs because the host is overutilized during the burst and the cycles needed for vMotion directly impact those available for VM use. For CloudPowerCap , there is a relatively small impact to performance after the burst and before DRS can run CloudPowerCap to reallocate the host power caps. The power cap setting operation itself can be executed by the host within 1 millisecond and introduce minor payload overhead.

5.5.3 Standby Host Power Reallocation

CloudPowerCap can reallocate standby hosts' power cap to increase the capacity of powered-on hosts and thereby their efficiency and ability to handle bursts. To demonstrate this, we consider the same initial setup in terms of hosts and VMs as in the previous experiment. In this case, all VMs are running a similar workload of 1.2 GHz and 2 GB memory demand. At time 750, each VM's demand reduces to 400 MHz, and when DRS is next invoked at time 900, DPM recommends that the VMs be consolidated onto two hosts and that the remaining host be powered-off. After the host has been evacuated and powered-off at time 1200, CloudPowerCap reassigns its power cap to 0 and reallocates the rack power budget to the two remaining hosts, setting their power caps to 320W each. At time 1400, there is an unexpected spike. In the case of statically-assigned power caps, the host that was powered-off is powered back on to handle the spike, but in the CloudPowerCap case, the additional CPU capacity available on the 2 remaining hosts given their 320 W power caps are sufficient to handle this spike and the powered-off host is not needed.

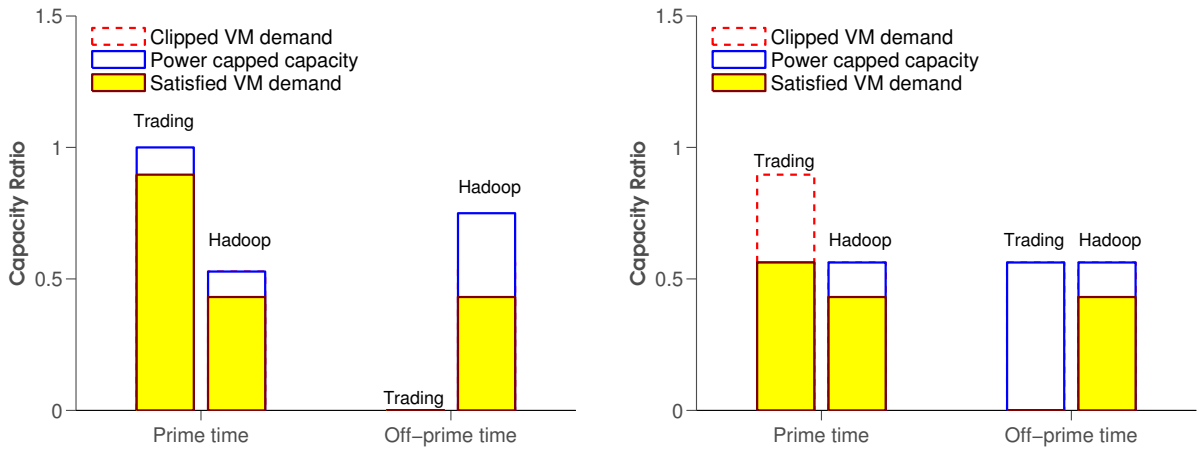
	CPU Payload Ratio	**vMotion**	**Power Ratio**
CPC	1.00	10	1.00
Static	0.98	19	1.36
StaticHigh	1.00	10	1.00

Table 5.4: CloudPowerCap (CPC) reallocating standby host power

Table 5.4 compares the CPU payload in cycles delivered to the VMs CloudPowerCap , Static and StaticHigh. In this case, a number of additional vMotions are needed for Static, but the overhead of these vMotions does not significantly impact the CPU payload, because there is plenty of headroom to handle this overhead. However, Static consumes much more power than the other 2 cases, since powering the additional host back on and repopulating it consumes significant power. In contrast, CloudPowerCap is able to match the power efficiency of the baseline, by being able to use peak capacity of the powered-on hosts.

5.5.4 Flexible Resource Capacity

CloudPowerCap supports flexible use of power to allow trade-offs between resource capacities to be made dynamically. To illustrate such a trade-off at scale, we consider a cluster of hosts as described in Section 2.1. The cluster is used to run both production trading VMs and production hadoop compute VMs. The trading VMs are configured with 2 vCPUs and 8 GB and they are idle half the day (off-prime time), and they run heavy workloads of 2x2.6 GHz and 7 GB demand the other half of the day (prime time). They access high-performance shared storage and hence are constrained to run on hosts with access to that storage, which is only mounted on 8 hosts in the cluster. The hadoop compute VMs are configured with 2 vCPUs and 16 GB and each runs a steady workload of 2x1.25 GHz and 14 GB demand. They access local storage and hence are constrained to run on their current hosts and cannot be vMotioned. During prime time, the 8 servers running the trading VMs do not receive tasks for the hadoop VMs running on those servers; this is accomplished via an elastic scheduling response to the reduced available resources [100]. Figure 5.8 shows the simulation results of the cluster under CloudPowerCap and Static configuration of power caps.



(a) CloudPowerCap (b) Static

Figure 5.8: Trade-offs between dynamical resource capacities. *Trading* indicates a group of servers running production trading while *Hadoop* represents servers run production Hapdoop.

Table 5.5 compares the CPU and memory payload delivered for three scenarios. The staticHigh scenario involves deploying 25 servers with power cpas of 320 W, which immediately and fully supports the trading VMs prime time demand but limits the overall available

memory and local disks in the cluster associated with the 25 servers. The Static senario instead involves deploying 32 servers with each host power cap statically set to 250 Watts. This scenario allows more memory and local disks to be accessed, increasing the overall CPU and memory payload delivered because more hadoop work can be accomplished, but limits the peak CPU capacity of each host, meaning that the trading VMs run at only 62 percent of their prime time demand. With CloudPowerCap, the benefits to the hadoop workload of the static scenario are retained, but the power caps of the hosts running the trading VMs can be dynamically increased, allowing those VMs' full prime time demand to be satisfied.

	CPU Payload Ratio	**Mem Ratio**	***Trading* Demand Ratio**
CPC	1.24	1.28	1.00
Static	1.21	1.28	0.62
StaticHigh	1.00	1.00	1.00

Table 5.5: CloudPowerCap (CPC) enabling flexible resource capacity

5.6 Related Work

Several research projects have considered power cap managment for virtualized infrastructure [53,72,73,78,98]. Among them, the research mostly related to our work is [73], in which authors proposed VPM tokens, an abstraction of changeable weights, to support power budgeting in virtualized environment. Like our work, VPM tokens enables shifting *power budget slack*, corresponding *headroom* in this paper, between hosts. However VPM tokens are independent to resource management system and may generate conflicting actions without coordination mechanisms.

Interoperating with a cloud resource management system like DRS also allows CloudPowerCap to support interesting additional features: 1) CloudPowerCap accomodates consolidation of physical servers caused by dynamic power management while previous work assumed a fixed working server set, 2) CloudPowerCap is able to handle and facilitate VM migration caused by correcting constraints imposed on physical servers and VMs, 3) CloudPowerCap can also deal with and enhance power cap management in the presence of load balancing which is not considered in the previous papers.

The authors of [78] describe managing performance and power management goals at server, enclosure, and data center level and propose handling the power cap hierarchically across multiple levels. Optimzation and feedback control algorithms are employed to coordinate the power management and performance indices for entire clusters. In [98], the authors build a framework to coordinate power and performance via Model Predictive Control through DVFS (Dynamic Voltage and Frequency Scaling). To provide power cap management through the VMs management layer, [72] proposed throttling VM CPU usage to respect the power cap. In their approach, feedback control is also used to enforce the power cap while maintaining system performance. Similarly, the authors in [53] also discussed data center level power cap management by throttling VM resource allocation. Like [78], they also adopted a hierachical approach to coordinate power cap and performance goals.

While all of these techniques attempt to manage both power and performance goals, their resource models for the performance goals are incomplete in various ways. For examples, none of the techniques support guaranteed SLAs (reservations) and fair share scheduling (shares). Some build a feedback model needing application-level performance metrics acquired from cooperative clients, which is rare especially in public clouds [7].

5.7 Summary

Many modern data centers have underutilized racks. Server vendors have recently introduced support for per-host power caps, which provide a server-enforced limit on the amount of power that the server can draw, improving rack utilization. However, this approach is tedious and inflexible because it needs involvement of human operators and does not adapt in accordance with workload variation. This paper presents CloudPowerCap to manage a cluster power budget for a virtualized infrastructure. In coordination with resource management, CloudPowerCap provides holistic and adaptive power budget management framework to support service level agreements, fairness in spare power allocation, entitlement balancing and constraint enforcement.

Chapter 6

Conclusion

This dissertation focus on development of dynamic thermal and power management of computer systems. This dissertation reserach comprises two parts. The first part develops dynamic thermal management for real-time systems running on different types of computing platforms. To meet the challenges posed by uncertainties in the thermal and power characteristics of computing systems, we employ feedback control-theoretic approaches to meet both the thermal and real-time performance requirements on single-core and multicore hosts.

To improve performance of thermal management on distributed real-time systems, an control-theoretic approach for real-time clusters is also developed in this part, which relies on thermal balancing rather than throttling to redistribute workload and reduce hot spots in the cluster.

The second part of the dissertation studies the problem of dynamic power cap management for cloud computing infrastructure. We developed, CloudPowerCap, a power cap management framework fro virtualized server clusters. The key of CloudPowerCap is to treat and manage the power cap in close *coordination* with resource management system so that no compromise on performance due to conflicting actions generated between CloudPowerCap and the resource management system.

References

[1] http://ark.intel.com/Product.aspx?id=27255.

[2] http://www.spec.org/.

[3] www.lm-sensor.org.

[4] Mehdi Amirijoo, Jörgen Hansson, Svante Gunnarsson, and Sang Hyuk Son. Quantifying and suppressing the measurement disturbance in feedback controlled real-time systems. *Real-Time Systems*, 40(1):44–76, 2008.

[5] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *COLP*, 2003.

[6] Anton Beloglazov and Rajkumar Buyya. OpenStack Neat: A Framework for Dynamic Consolidation of Virtual Machines in OpenStack Clouds - A Blueprint. Technical report, CLOUDS-TR-2012-4, The University of Melbourne, 2012.

[7] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The resource-as-a-service (RaaS) cloud. In *HotCloud'12*, 2012.

[8] DP Bertsekas, RG Gallager, and P Humblet. *Data networks*. Prentice-Hall, 1992.

[9] A. Block, B. Brandenburg, J. Anderson, and S. Quint. Adaptive multiprocessor real-time scheduling with feedback control. In *ECRTS*, July 2009.

[10] S. Brandt and G. J. Nutt. Flexible soft real-time processing in middleware. *Real-Time System*, 22(1-2), 2002.

[11] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *HPCA*, 2001.

[12] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Arzen. Feedback feedforward scheduling of control tasks. *Real-Time System*, 23(1-2):25–53, 2002.

[13] Thidapat Chantem, Robert P. Dick, and X. Sharon Hu. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. In *DATE*, 2008.

[14] Jian-Jia Chen, , Shengquan Wang, and Lothar Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *RTAS*, 2009.

[15] Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In *RTAS*, 2007.

[16] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In *ISLPED*, pages 213–218, New York, NY, USA, 2007. ACM.

[17] Ayse Kivilcim Coskun, Tajana Šimunic Rosing, and Kenny C. Gross. Utilizing predictors for efficient thermal management in multiprocessor socs. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1503–1516, October 2009.

[18] JD Davis, S Rivoire, and M Goldszmidt. Star-Cap: Cluster Power Management Using Software-Only Models. Technical report, MSR-TR-2012-107, Microsoft Research, 2012.

[19] Dell Inc. Dell Energy Smart Management. 2012.

[20] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. *SIGARCH Computer Architecture News*, 34(2):78–88, 2006.

[21] EPA. United states environmental protection agency. report to congress on server and data center energy efficiency., 2007.

[22] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *SIGARCH Comput. Archit. News*, 35(2):13–23, June 2007.

[23] Alexandre P Ferreira, Daniel Mosse, and Jae C. Oh. Thermal faults modeling using a RC model with an application to Web farms. In *ECRTS*, 2007.

[24] N. Fisher, Jian-Jia Chen, Shengquan Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *RTAS*, 2009.

[25] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems*. Addison Wesley Longman, Inc., 1998.

[26] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems(Third Editon)*. Addison Wesley Longman, Inc, Menlo Park, CA, 1998.

[27] Xing Fu, Xiaorui Wang, and Eric Puster. Dynamic thermal and timeliness guarantees for distributed real-time embedded systems. In *RTCSA*, 2009.

[28] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D. Koutsoukos, and Hongan Wang. Feedback Thermal Control for Real-time Systems. In *RTAS*, 2010.

[29] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D. Koutsoukos, and Hongan Wang. Feedback Thermal Control of Real-time Systems on Multicore Processors. Technical Report WUCSE-2011-3, Washington University in St. Louis, 2011.

[30] Gartner Research. Shrinking Data Centers: Your Next Data Center Will Be Smaller Than You Think. Mar 2011.

[31] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. *SIGOPS Operating System Review*, 38(5):260–270, 2004.

[32] G. Grimm, M.J. Messina, S.E. Tuna, and A.R. Teel. Nominally robust model predictive control with state constraints. *IEEE Transactions on Automatic Control*, 52(10):1856–1870, 2007.

[33] G. Grimm, A.R. Teel, and L. Zaccarian. The l2 anti-windup problem for discrete-time linear systems: Definition and solutions. *Systems & Control Letters*, 57(4):356–364, 2008.

[34] A Gulati, G Shanmuganathan, A Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *HotCloud*, 2011.

[35] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. VMware Distributed Resource Management: Design, Implementation, and Lessons Learned. *VMware Technical Journal*, Mar 2012.

[36] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

[37] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, Yogesh Jaluria, and Ricardo Bianchini. Mercury and freon: temperature emulation and management for server systems. In *ASPLOS*, 2006.

[38] G. Herrmann, M.C. Turner, and I. Postlethwaite. Discrete-time and sampled-data anti-windup synthesis: stability and performance. *International Journal of Systems Science*, 37(2):91–113, 2006.

[39] HP Inc. HP Power Capping and HP Dynamic Power Capping for Proliant Servers. 2011.

[40] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. A framework for dynamic energy efficiency and temperature management. In *MICRO*, 2000.

[41] Wei Huang, Mircea R. Stan, Kevin Skadron, Karthik Sankaranarayanan, Shougata Ghosh, and Sivakumar Velusam. Compact thermal modeling for temperature-aware design. In *DAC*, 2004.

[42] W. Hung, Y. Xie, N. ViJáykrishnan, M. Kandemir, and M. J. Irwin. Thermal-aware task allocation and scheduling for embedded systems. In *DATE*, pages 898–899, 2005.

[43] IBM Inc. IBM Active Energy Manager. 2012.

[44] Intel Corp. Intel Pentium 4 processor in the 423-pin package thermal design guidelines. Technical report, Intel, 2000.

[45] Canturk Isci and Margaret Martonosi. Identifying program power phase behavior using power vectors. In *Proceedings of IEEE International Workshop on Workload Characterization*, 2003.

[46] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO*, 2003.

[47] Kyoung-Don Kang, Jisu Oh, and Sang Hyuk Son. Chronos: Feedback control of a real database system performance. In *RTSS*, 2007.

[48] Nicholas Kottenstette, Joseph Hall, Xenofon Koutsoukos, Panos Antsaklis, and Janos Sztipanovits. Digital control of multiple discrete passive plants over networks. *International Journal of Systems, Control and Communications (IJSCC)*, Special Issue on Progress in Networked Control Systems(2):194–228, 2011.

[49] Nicholas Kottenstette, Heath LeBlanc, Emeka Eyisi, and Xenofon Koutsoukos. Multirate networked control of conic (dissiptive) systems. In *American Control Conference*, 2011.

[50] M. Lazar, D. Muñoz de la Peña, W. Heemels, and T. Alamo. On input-to-state stability of min–max nonlinear model predictive control. *Systems & Control Letters*, 57(1):39–48, 2008.

[51] Jong Sung Lee, K. Skadron, and Sung Woo Chung. Predictive temperature-aware DVFS. *IEEE Transactions on Computers*, 59(1):127–133, 2010.

[52] Weiping Liao, Lei He, and K. M. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *IEEE Transaction Computer-Aided Design Integrated Circuits System*, 24(7):1042–1053, 2005.

[53] H Lim, A Kansal, and J Liu. Power Budgeting for Virtualized Data Centers. In *USENIX ATC*, 2011.

[54] Mikael Lindberg and Karl-Erik Årzén. Feedback control of cyber-physical systems with multi resource dependencies and model uncertainties. In *RTSS*, 2010.

[55] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1:46–61, January 1973.

[56] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[57] J.W.S. Liu. *Real-Time systems*. Prentice Hall, 2000.

[58] Yongpan Liu, Huazhong Yang, Robert P. Dick, Hui Wang, and Li Shang. Thermal vs Energy Optimization for DVFS-Enabled Processors in Embedded Systems. In *ISQED*, 2007.

[59] C. Løvaas, M.M. Seron, and G.C. Goodwin. Robust output-feedback model predictive control for systems with unstructured uncertainty. *Automatica*, 44(8):1933–1943, 2008.

[60] C. Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-time Systems*, 23, 2002.

[61] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, 2005.

[62] J. M. Maciejowski. *Predictive Control with Constraints*. Pearson Eduction Limited, Edinburg Gate, England, 2002.

[63] D.Q. Mayne, J.B. Rawlings, C.V. Rao, and PO Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36:789–814, 2000.

[64] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *SIGOPS Operating System Review*, 40(4):403–414, 2006.

[65] Richard H. Middleton and Graham C. Goodwin. *Digital Control and Estimation: A Unified Approach*. Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1990.

[66] Lauri Minas and Brad Elison. The problem of power consumption in servers, 2009. "www.intel.com".

[67] J. Moore, R. Sharma, R. Shih, J. Chase, C. Patel, and P. Ranganathan. Going beyond CPUs: The portential of temperature-aware solutions for the data center. In *TACS*, June 2004.

[68] Justin Moore, Jeff Chase, and Parthsarathy Ranganathan. Weatherman: Automated, online, and predictive thermal mapping and management for data centers. In *ICAC*, June 2006.

[69] F Mulas, D Atienza, A Acquaviva, S Carta, L Benini, and G De Micheli. Thermal Balancing Policy for Multiprocessor Stream Computing Platforms. 28(12):1870–1882, 2009.

[70] Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd, Luca Benini, and Giovanni De Micheli. Temperature control of high-performance multi-core platforms using convex optimization. In *DATE*, 2008.

[71] A. Mutapcic, S. Boyd, S. Murali, D. Atienza, G. De Micheli, and R. Gupta. Processor speed control with thermal constraints. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 56(9):1994–2008, 2009.

[72] R. Nathuji, P. England, P. Sharma, and A. Singh. Feedback driven QoS-aware power budgeting for virtualized servers. In *FeBID*, 2009.

[73] R Nathuji, K Schwan, A Somani, and Y Joshi. VPM tokens: virtual machine-aware power budgeting in datacenters. *Cluster computing*, 2009.

[74] OpenStack. Openstack compute administration manual.

[75] A.V. Oppenheim, A.S. Willsky, and S.H. Nawab. *Signals and systems*. Prentice hall Upper Saddle River, NJ, 1997.

[76] R. Ortega, A. J. Van Der Schaft, I. Mareels, B. Maschke, and Lss G. Y. Supelec. Putting energy back in control. *Control Systems Magazine, IEEE*, 21(2):18–33, 2001.

[77] Gang Quan and Yan Zhang. Leakage aware feasibility analysis for temperature-constrained hard real-time periodic tasks. In *ECRTS*, 2009.

[78] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 48–59. ACM, 2008.

[79] Ravishankar Rao, Sarma B. K. Vrudhula, and Chaitali Chakrabarti. Throughput of multi-core processors under thermal constraints. In *ISLPED*, 2007.

[80] Rodrigo M. Santos, Giuseppe Lipari, and Enrico Bini. Efficient on-line schedulability test for feedback scheduling of soft real-time tasks under fixed-priority. In *RTAS*, 2008.

[81] 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, 2002.

[82] Jayanth Srinivasan and Sarita V. Adve. Predictive dynamic thermal management for multimedia applications. In *ICS*, June 2003.

[83] John A. Stankovic, Tian He, Tarek Abdelzaher, Mike Marley, Gang Tao, Sang Son, and Chengyang Lu. Feedback control scheduling in distributed real-time systems. In *RTSS*, Dec. 2002.

[84] Anja Strunk. Costs of Virtual Machine Live Migration: A Survey. In *IEEE Eighth World Congress on Services*, 2012.

[85] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2:437–445, 1994.

[86] Tom's Hardware. 3.8 GHz P4-570 and e0 stepping to end Intel's performance crisis. Technical report, Tom's Hardware, 2004.

[87] A. J. van der Schaft. *L2-Gain and Passivity in Nonlinear Control*. New York:Springer-Verlag, 1999.

[88] VMware, Inc. Resource Management with VMware DRS, 2006.

[89] VMware, Inc. VMware Distributed Power Management Concepts and Use, 2010.

[90] VMware, Inc. VMware vSphere vMotion: Architecture, Performance, and Best Practices in VMware vSphere 5, 2011.

[91] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI*. USENIX Association, 1994.

[92] Shengquan Wang and Riccardo Bettati. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *RTSS*, 2006.

[93] Shengquan Wang and Riccardo Bettati. Reactive speed control in temperature-constrained real-time systems. In *ECRTS*, 2006.

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

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

[96] X. Wang, Y. Chen, C. Lu, and X. Koutsoukos. Towards controllable distributed real-time systems with feasible utilization control. *IEEE Transactions on Computers*, 58(8):1095–1110, August 2009.

[97] X. Wang, C. Lu, and C. Gill. Fcs/norb: A feedback control real-time scheduling service for embedded orb middleware. *Microprocessors and Microsystems*, 32(8):413–424, 2008.

[98] Xiaorui Wang and Yefu Wang. Coordinating Power Control and Performance Management for Virtualized Server Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 22(2):245–259, February 2011.

[99] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. *SIGARCH Computer Architecture News*, 37(3):314–324, 2009.

[100] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[101] Fetahi Wuhib, Rolf Stadler, and Hans Lindgren. Dynamic resource allocation with management objectives - Implementation for an OpenStack cloud. In *CNSM*. IEEE, 2012.

[102] L. Yuan, S. Leventhal, and G. Qu. Temperature-aware leakage minimization techniques for real-time systems. In *ICCAD*, November 2006.

[103] E. Zafiriou. Robust model predictive control of processes with hard constraints. *Computing Chemical Engineering*, 14(4):359–371, 1990.

[104] F. Zanini, D. Atienza, and G. De Micheli. A control theory approach for thermal balancing of MPSoC. In *ASP-DAC*, 2009.

[105] F. Zanini, C. N. Jones, D. Atienza, and G. De Micheli. Multicore thermal management using approximate explicit Model Predictive Control. In *ISCAS*, 2010.

[106] Francesco Zanini, David Atienza, Luca Benini, and Giovanni De Micheli. Multicore Thermal Management with Model Predictive Control. In *ECCTD*, 2009.

[107] A. Zheng. Robust stability analysis of constrained model predictive control. *Journal of Process Control*, 9(4):271–278, 1999.

[108] Yifan Zhu and Frank Mueller. Dvsleak: combining leakage reduction and voltage scaling in feedback edf scheduling. In *LCTES*, 2007.