UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

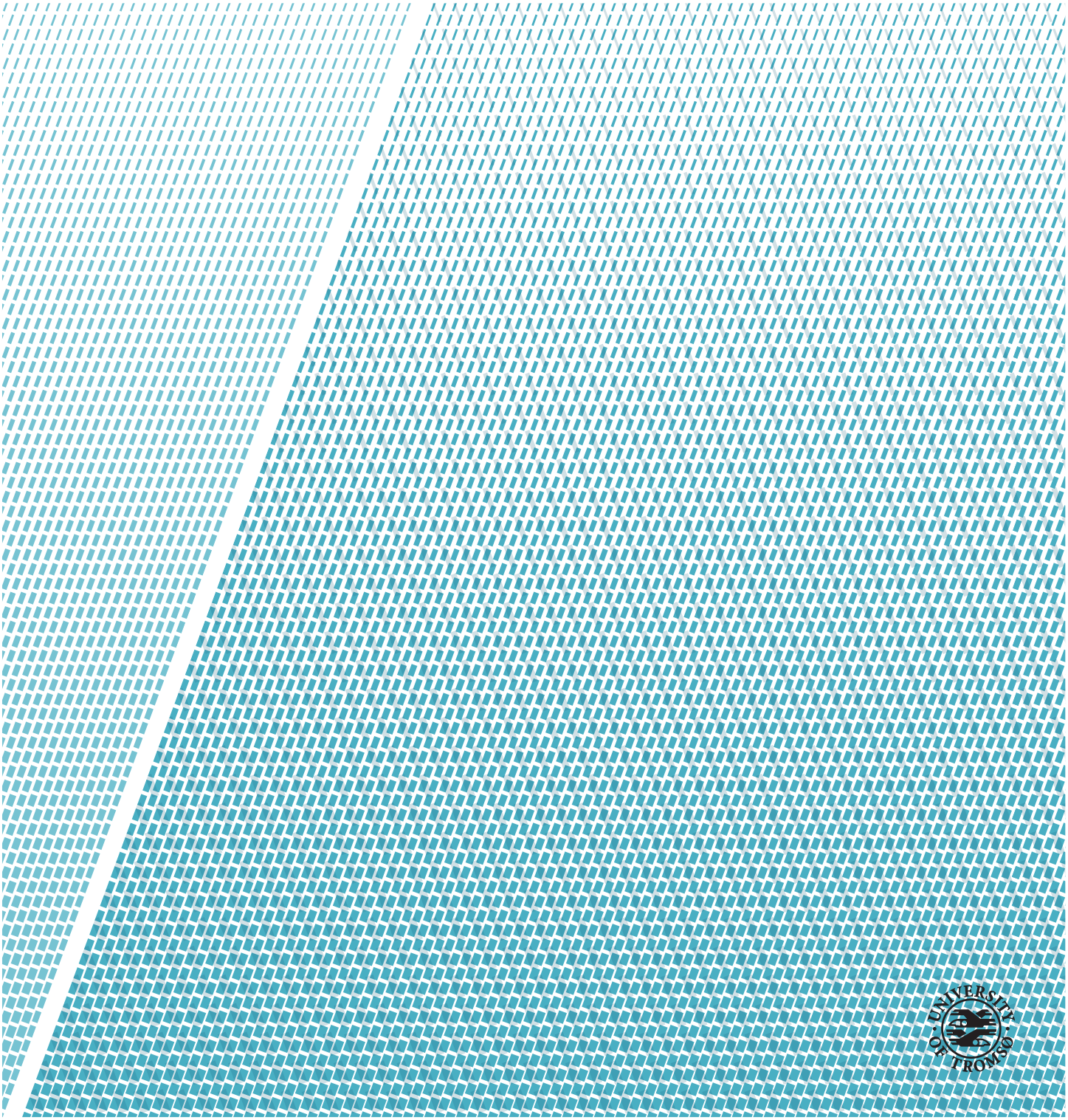
ROPE: Reducing the Omni-kernel Power Expenses

Implementing power management in the Omni-kernel architecture

—

Jan-Ove A. Karlberg

INF-3990 Master thesis in Computer Science, May 2014



“That’s it man, game over man, game over!
What the fuck are we gonna do now?
What are we gonna do?”
–Hudson, Aliens (1986)

Abstract

Over the last decade, power consumption and energy efficiency have arisen as important performance metrics for data center computing systems hosting cloud services. The incentives for reducing power consumption are several, and often span economic, technological, and environmental dimensions. Because of the vast number of computers currently employed in data centers, the economy of scale dictates that even small reductions in power expenditure on machine level can amount to large energy savings on data center scale.

Clouds commonly employ hardware virtualization technologies to allow for higher degrees of utilization of the physical hardware. The workloads encapsulated by virtual machines constantly compete for the available physical hardware resources of their host machines. To prevent execution of one workload from seizing resources that are intended for another, absolute visibility and opportunity for resource allocation is necessary. The Omni-kernel architecture is a novel operating system architecture especially designed for pervasive monitoring and scheduling. Vortex is an experimental implementation this architecture.

This thesis describes ROPE (Reducing the Omni-kernel Power Expenses), which introduces power management functionality to the Vortex implementation of the Omni-kernel architecture. ROPE reduces the power consumption of Vortex, and does so while limiting performance degradation.

We evaluate the energy savings and performance impacts of deploying ROPE using both custom tailored and industry standard benchmarks. We also discuss the implications of the Omni-kernel architecture with regards to power management, and how energy efficiency can be accommodated in this architecture.

Acknowledgements

I would like to thank my supervisor, Dr. Åge Kvalnes for his great advice, ideas, and highly valued input throughout this project. Your passion and knowledge is unmatched. The realization of this project would not have been possible without you.

Further I would like to thank Robert Pettersen for his invaluable help. Your incredible insight into the Omni-kernel architecture project has been a life-saver in many a situation. Many thanks for putting up with all my requests for new convenience tools.

I would also like to thank some of my fellow students whose help have been crucial throughout this project. In no particular order: Kristian Elsebø, thank you so much for all help with diskless servers, HammerDB, and general sysadmin stuff. Erlend Graff, your knowledge, kindness, and enthusiasm is simply incredible. Only he who has seen the red zone can understand my gratitude. Einar Holsbø, my harshest literary critic: thank you so much for your valued advice, and for proof-reading this entire text. To all of the above, and all the rest of you whose names are to many to list here: Thank you so much, I am truly blessed to be able to call you my friends.

To my family, parents and brothers: without your continuous support none of this would have been possible.

Finally, to my girlfriend Martine Espeseth. Thank you so much for all your love and support. Especially through the last month's coffee fueled frenzy.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

μ-ops micro-ops

AB ApacheBench

ABI application binary interface

ACPI Advanced Configuration and Power Interface

ACPICA ACPI Component Architecture

AML ACPI Machine Language

APM advanced power management

CDN content delivery network

CMOS Complementary metal-oxide-semiconductor

CPU central processor unit

CPUID cpu Identification

DC data center

DPM dynamic power management

DRR dynamic round-robin

DVFS dynamic voltage and frequency scaling

DVS dynamic voltage scaling

EM enforcement mechanism

EMA exponential moving average

FFH functional fixed hardware

GPHR Global Phase History Register

GPHT Global Phase History Table

HTTP HyperText Transfer Protocol

I/O input/output

ICT information and communication technologies

IPMI Intelligent Platform Management Interface

LKM loadable kernel module

LLC last-level cache

LRU least recently used

MASE mean absolute scaled error

MWAIT Monitor Wait

NOPM new-order transactions per minute

OKA omni-kernel architecture

OLTP online transaction processing

OS operating system

OSL OS Service Layer

OSPM Operating System-directed Power Management

PCI Periperhal Component Interconnect

PCIe Peripheral Component Interconnect Express

PHT Pattern History Table

PM power management

PMC performance monitoring counter

PMI performance monitoring interrupt

PSM Policy State Manager

QoS quality of service

RLE run-length encoding

ROPE Reducing the Omni-kernel Power Expenses

SLA service level agreement

SLO service level objective

SMA simple moving average

SoC system on chip

TPC Transaction Processing Performance Council

TPM transactions per minute

URL Uniform Resource Locator

VFI voltage/frequency island

VLAN Virtual LAN

VM virtual machine

VMM virtual machine monitor

VPC Virtual Private Cloud

WLAN wireless local area network

1 Introduction

Over the last decade, power consumption and energy efficiency have arisen as important performance metrics for data center (DC) computing systems hosting cloud services. The incentives for reducing power consumption are several, and often span economic, technological, environmental, and even political dimensions. Already in the mid nineties, the concept of "green computers" surfaced as a response to increased CO_2 emissions and growing costs [85]. Emissions and costs have only increased with time, and in 2007, the information and communication technologies (ICT) sector was estimated to have been responsible for approximately 2% of all global carbon emissions, 14% of this being attributable to data center operation [87]. In 2012, Facebook's DCs alone had a carbon footprint of 298 000 tons of CO_2 [8], roughly the same as emissions originating from the electricity use of 24 000 homes [1].

Emissions originating from DCs are expected to double by 2020 [87]. Thus, DC energy consumption is no longer only a business concern, but also a national energy security and policy concern [90]. For example, in the United States, the U.S. Congress is taking the issue of DC energy efficiency to the national level [4]. Equivalently, the European Commission is considering the introduction of a voluntary code of conduct regarding energy efficiency in DCs [5]. It is expected that other regions and countries will introduce similar regulations and legislation within the foreseeable future [90].

Technologically, reducing power consumption is important mainly for two reasons. First, high power consumption is directly correlated with increased heat

dissipation. As computers are getting both smaller and packed more densely, the necessary cooling solutions become increasingly expensive. For instance, a cooling system for a 30.000 square feet, 10MW data center can cost $2–$5 million [71]. According to to the same source, every watt spent by computing equipment necessitates the consumption of an additional 0.5 to 1.0W operating the cooling infrastructure. This amounts to $4–$8 million in annual operating costs [76]. In data centers, where availability is essential, failures within the cooling infrastructure can be a major problem as this can result in reduced mean time between failure, and even service outages [64].

Towards the turn of the millennium, huge data centers started emerging across the globe. With as many as tens of thousands of servers [64], energy consumption again surfaced as a key design constraint. For some facilities the electricity expenses are one of the dominant costs of operating the DC, accounting for 10% of the total cost of ownership [49]. Although the rate at which power consumption in DCs increase is slowing down [48], trends of increased energy consumption in DCs are likely to continue.

Because of the vast number of computers currently employed in DCs, the economy of scale dictates that even small reductions in power expenditure on machine and device level can amount to large energy savings on data center scale.

At the machine level, clouds commonly employ hardware virtualization technologies to allow for higher degrees of utilization of the physical hardware. However, this approach is not without problems. The workloads encapsulated by virtual machines constantly compete for the available physical hardware resources of their host machines, and interference from this resource sharing can cause unpredictable performance. Even when virtual machine (VM) technologies are employed, some operating system (OS) will have to serve as a foundation, enabling the allocation of resources to the VMs. Virtual machine monitor (VMM) functionality is implemented as an extension to such an operating system.

The omni-kernel architecture (OKA) is a novel operating system architecture especially designed for pervasive monitoring and scheduling [52], [53]. The architecture ensures that all resource consumption is measured, that the resource consumption resulting from a scheduling decision is attributed to the correct activity, and that scheduling decisions are fine-grained.

Vortex [52], [53], [68], [69], [67] is an experimental implementation of the OKA providing a novel and light weight approach to virtualization. While many conventional VMMs expose virtualized device interfaces to its guest VMs [11], [78, p. 254–258], Vortex offers high-level commodity OS abstrac-

tions. Examples of such abstractions are files, memory mappings, network connections, processes, and threads. By doing this, Vortex aims to offload common OS functionality from its guest OSs, reducing both the resource footprint of a single VM and the duplication of functionality across all the VM OSs. Vortex does not currently provide a complete virtualization environment capable of hosting ports of commodity OSs. Instead, a light weight approach that targets compatibility at the application level is used; thin VM OSs similar to compatibility layers emulate the functionality of existing OSs, retaining their application binary interfaces (ABIs). Currently, a VM OS capable of running Linux applications such as Apache[1], MySQL[2], and Hadoop[3] exists [68], [69], while efforts to do the same for Windows applications are under way [36].

This thesis describes ROPE (Reducing the Omni-kernel Power Expenses), which introduces several dynamic power management algorithms to the Vortex implementation of the OKA.

1.1 Problem Definition

This thesis shall design, deploy, and evaluate power management functionality within the Vortex operating system kernel. Focus will be to conserve energy while limiting performance degradation.

1.2 Scope and Limitations

Power management is a multifaceted problem. In the context of data centers, a multitude of approaches and techniques are currently being employed. In order to save as much energy as possible, all aspects of power consumption must be analyzed, understood, and managed cleverly. In this thesis, the focus lies on power management on a per-node basis. We do not focus on techniques involving multiple nodes, for instance through consolidation of virtual machines [55], [43], [81], intelligent placement of workloads [32], [64], or scheduling of work based on the availability of green power [10], [13]. Further, we do not focus on the use of exotic or mobile technology in order to save power [59], [34]. Rather, we specifically target software techniques for power management in standard enterprise blade servers.

1. http://httpd.apache.org/
2. http://www.mysql.com/
3. http://hadoop.apache.org/

1.3 Interpretation

Our problem concerns effectuating PM while limiting performance degradations. If tradeoffs between energy savings and performance must be made, the evaluation of the implemented functionality should enable a user to make enlightened decisions. This requires that the system be tested with relevant benchmarks and workloads. Specifically, the implemented system should adhere to the following principles and requirements:

Design Goals and Principles

- **Simplicity:** Where possible, simple solutions should be adopted. This is especially important as Vortex is under continuous development by faculty and students.
- **Flexibility:** Flexibility is preferable over rigidity: Vortex is constantly changing. If ROPE is to remain relevant its implementation must be both flexible and robust.
- **Orthogonality:** ROPE should target the issues of energy efficiency along orthogonal axes. Solutions and policies should compliment rather than rely on each other, making future additions and modifications as uncomplicated as possible.

Requirements

- The system should reduce the power consumption of Vortex.
- The system should make use of existing standards for power management (PM) wherever possible.
- The implementation of the system should focus on performance. For all implemented functionality, performance implications should be evaluated.

1.4 Methodology

According to the final report of the ACM Task Force on the Core of Computer Science, the discipline of computing can be divided into three major paradigms. These paradigms are theory, abstraction, and design.

The first paradigm is *theory*. Rooted in mathematics, the theoretical approach is characterized by first defining problems, then proposing theorems, and finally seeking to prove them in order to determine potentially new relationships and make progress in computing.

Abstraction is rooted in the experimental scientific method. Following this approach one seeks to investigate a phenomenon by constructing a hypothesis, and then make models or perform simulations to challenge this hypothesis. Finally, the results are analyzed.

Design is the last paradigm, and is rooted in engineering. Within this paradigm, one seeks to construct a system for solving a given problem. First, the requirements and specifications of said system are defined. The system should then be designed and implemented according to these requirements and specifications. Upon completion of the system, it should be tested and evaluated.

For this thesis, the design paradigm seems to be the most fitting. We have stated a problem, its requirements, and specifications. Now, a prototype solving the problem according to the specification must be designed and implemented. The testing of the system will amount to quantitatively evaluating its performance and cost of operation.

1.5 Context

This project is written as a part of the Information Access Disruption (iAD) center. The iAD center focuses on research into fundamental concepts and structures for large-scale information access. The main focus areas are technologies related to sport, cloud computing, and analytic runtimes.

Specifically, this project is related to the Omni-kernel architecture project, where a novel OS architecture providing total control over resource allocation is under development. It is in Vortex, the current instantiation of this architecture, that ROPE has been implemented.

1.6 Outline

Chapter 2 covers some basics of power management, introduces some key concepts, and outlines often employed classes of PM algorithms. It also introduces different power management mechanisms.

Chapter 3 describes the architecture and design of ROPE, and how power management functionality have been introduced to the Vortex operating system kernel. Further, this chapter explains how different PM policies coexist and complement each other.

Chapter 4 covers the implementation specific details of the PM policies described in chapter 3. It also covers ACPI and how it is incorporated into the Vortex kernel. Intel-specific PM capabilities are also discussed.

Chapter 5 evaluates the power management policies implemented in ROPE, and contains our experiments and results.

Chapter 6 presents related work.

Chapter 7 discuss some of our findings and results in greater detail, and covers how energy efficiency can be accommodated in the Omni-kernel architecture. Finally, it concludes this thesis.

2 Power Management

This chapter covers the basic principles of power management (PM). First, key terms and concepts are introduced. Following this, different approaches to PM as well as common classes of algorithms are outlined.

2.1 Introduction to Power Management

Dynamic power management (DPM) [70], [19], [77] is a design methodology for dynamic configuration of systems, components, or software, that aims to minimize the power expenditure of these under the current workload, and constraints of service and performance levels requested by the user.

DPM can be applied by different techniques that selectively deactivate or power down components when they are idle or unexploited. The goal of all such techniques is to achieve more energy efficient computation, and a central concept of DPM is the ability to trade off performance for power savings in a controlled fashion while also taking the workload into account. The main function of a power management scheme, or policy, is to determine *when* to initiate transitions between component power states, and to decide *which* transitions to perform. For instance, if a device supports more than one PM state, one of these states may be more suitable to reduce energy consumption under a specific workload, than another. Consider, if the workload is memory bound, it might be possible to lower the frequency of a central processor unit

(CPU), thereby reducing power consumption, while still managing to perform all computations in a timely fashion.

Exactly what decision is (or should be) made at any time will be highly dependent on the active PM policy. Any performance constraints will play a fundamental role when making power management decisions [14], [70], regardless of whether the policy makes use of historical-, current-, or predicted future workloads to find the best course of action.

2.2 Evaluating Power Management Policies

Throughout this text, the term *workload*, as defined in [41], describes the physical usage of the system over time. This usage can be measured in any unit suitable as a metric. For instance, when implementing a policy that will power up or down a network interface card, average bandwidth and latency might be suitable metrics for capturing the characteristics of the workload. When deciding to spin down a hard drive, the number of read and write operations might be of more interest. Often, consecutive measurements of the workload characteristics are stored in a *trace*.

When attempting to conserve power by means of DPM policies, a problem that needs to be considered is the possibly detrimental effects of said policies on system performance. More precisely, it is desirable that the negative impact on metrics such as latency and job throughput is reduced to the extent possible [65], [70], [19], [77]. Because this is a critical issue, a means of comparing the amount of performance degradation of different policies or algorithms for dynamic power management is necessary. One way of approaching this is to define some baseline that different policies and algorithms can be compared to.

An important consideration of any power management system is that transitions between different power states of a device most often comes with a cost. If a transition is not instantaneous, and if the device is non-operational during this transition, performance can be lost when a transition is performed. Whenever transitions are non-instantaneous, there might also be a power cost associated with the transition. If the costs in performance, power, or both, are large, a power state might become close to useless as amortizing the transition costs becomes difficult [14].

Given a trace corresponding to some workload, an optimal baseline can be created in hindsight. Note that because no system can be all knowing of past, present, and future workloads, the optimal baseline can only be generated

offline. We define this optimal baseline, an *all knowing oracle*, as having the following characteristics:

- The device should not be powered down if this will affect performance negatively.
- The device should not change power states if the state transition will result in a reduction of saved power, i.e., the state transition will consume more power than is saved by entering the state and remaining in it throughout the idle period.
- The device should be powered down at all points in time not covered by the above cases, thereby maximizing the amount of saved power.

In [14], the authors survey several different approaches to DPM. We will adopt their notation when explaining applicability of dynamic power management. When switching a device into an inactive (sleep) state, this results in a period of inactivity. We denote the duration of a generic period at time n, T_n. This is the sum of the time spent in a state S, as well as the time spent transitioning into and out of the state. The *break-even time* for a given state, S, is denoted $T_{BE,S}$, and corresponds to the amount of time that must be spent in the state S in order to compensate for the power spent transitioning to and from it. Most often, lower power states have higher associated performance penalties. In the case that $T_n < T_{BE,S}$, there is typically not enough time to both enter and leave the inactive (sleep) state, or the time spent in the low-power state is not long enough to amortize the cost of transitioning to and from it. This means that if all idle periods corresponding to a given workload are shorter than $T_{BE,S}$, this power state can not be used to save any power. Transitions into a low-power state during an idle period that is shorter than the break-even time for this state is guaranteed to waste energy. Because of this, it is often desirable to filter out idle times of very short duration [84].

T_{BE} is the sum of two terms: $T_{BE} = T_{TR} + P_{TR}$. The first of these, T_{TR}, is the *total transition time*, meaning the time necessary to both enter and leave the low-power state. The second, *transition power* (P_{TR}), is the time that must be spent in the low-power state to compensate for the power spent during the transition phases. T_{TR} and P_{TR} are given by equations 2.1 and 2.2.

$$T_{TR} = T_{\text{On,Off}} + T_{\text{Off,On}} \tag{2.1}$$

Where $T_{\text{On,Off}}$ and $T_{\text{Off,On}}$ are the times necessary to perform a transition from

the on- to the off-state, and from the off- to the on-state respectively.

$$P_{TR} = \frac{T_{\text{On,Off}} P_{\text{On,Off}} + T_{\text{Off,On}} P_{\text{Off,On}}}{T_{TR}} \tag{2.2}$$

Using equation 2.1 and 2.2, T_{BE} can be expressed in the following manner:

$$T_{BE} = \begin{cases} T_{TR} + T_{TR} \frac{P_{TR} - P_{\text{On}}}{P_{\text{On}} - P_{\text{Off}}} & \text{, if } P_{TR} > P_{\text{On}} \\ T_{TR} & \text{, if } P_{TR} \leq P_{\text{On}} \end{cases} \tag{2.3}$$

Further, the energy saved by entering a state S for an idle period T_{idle}, with a duration longer than the break even time for this state, $T_{BE,S}$ is given by the following equation:

$$E_S(T_{\text{idle}}) = (T_{\text{idle}} - T_{TR})(P_{\text{On}} - P_S) + T_{TR}(P_{\text{On}} - P_{TR}) \tag{2.4}$$

In [35], the authors introduce two concepts useful when evaluating the quality of power management policies. The first of these—*external measures*—gives a quantitative measure of the interference between the PM policy and other applications. Any profit resulting from the use of the policy will also be quantified with external measures. Examples of metrics covered by external measures are power consumption, latency, and system resource consumption. *Internal measures*, on the other hand, can be used to quantify the accuracy of prediction algorithms. While the internal measures are useful when deciding on which prediction algorithms to use, the external measures are the ones that really matter for a running system.

2.3 Approaches to Power Management

In general, approaches to PM are either *heuristic* or *stochastic*. Although their goals are similar, the two classes of policies differ widely in their implementation as well as their theoretical foundations. Stochastic policies are theoretically optimal, and use statistical distributions to model workloads. Heuristic approaches, on the other hand, are not theoretically optimal, but often based on experimental results and practical engineering. Because of this, heuristic approaches such as static timeout policies—albeit often used with great

success—may lead to bad performance and unnecessary power consumption when the request rate for a service is highly variable [77].

PM policies can be further divided into discrete-time or event-driven variants. With discrete-time policies, decisions are re-evaluated at every time slice. Event-driven policies, on the other hand, only evaluate what action to take when some event, such as the arrival or completion of a request, occurs. The fact that discrete time solutions continuously re-evaluate the power settings of the system means that power is wasted if the device could otherwise be placed in a lower power state [84]. Further, PM policies can be split into stationary and non stationary policies. A stationary policy is constant in that it remains the same for all points in time, while a non stationary policy changes over time according to the experienced workload.

2.3.1 Heuristic Power Management Policies

Heuristic PM algorithms can in broad terms be split into two classes: timeout- and rate-based. Common for all heuristic policies, is that results are not guaranteed to be optimal.

Timeout-based Policies

The most common heuristic policies are variants of timeout policies [84]. Whenever an idle period occurs, a timer is started. A state transition is initiated if the idle period is longer than some pre-defined timeout value. Timeout based policies have the disadvantage that they result in unnecessary power expenditure while waiting for the timeout to expire [85]. A static timeout based policy always employs the same timeout value, whereas adaptive timeout policies modify their timeout parameters in order to better meet some performance goal [35]. Simple adaptive policies can result in larger power savings for a given level of latency [39].

Rate-based Policies

With rate-based policies, some prediction unit or algorithm calculates an estimate (prediction) of future load for the power-managed device or system. Typical examples are estimation of idle periods [35], the number of incoming requests, or the bandwidth consumption of a webserver. Rate-based policies make the system or component perform a state transition as soon as an idle period begins, given that it is likely that the idle period will be long enough to save power [85]. The efficiency of an algorithm therefore depends on the

ability to estimate the duration of the idle periods to a satisfactory degree, as failure to do so can cause performance penalties and also result in waste of energy. Normally, thresholds are used to determine when to power a device up or down. Examples of rate-based predictors are different variants of moving averages.

2.3.2 Stochastic Power Management Policies

Stochastic approaches to PM differ from heuristic techniques in that they guarantee optimal results as long as the system to be managed is modeled well enough [84]. All stochastic models use statistical distributions to describe the system that is to be power managed. For instance, while one distribution describes the time spent transitioning to and from different power states, another might describe the time spent by some resource to service a client's request, while a third might model the inter-arrival times of these requests. The distribution used to model the different parts of the system varies, and one typically tries to find a distribution that closely matches the real system. Although theoretically optimal, the quality of these models are highly dependent on the system being sufficiently well-modeled. Further, as the optimal solution is calculated from traces, the model quality will always be constrained by the trace quality, and the degree to which distributions fit the actual usage patterns [75].

2.3.3 Machine Learning Based Power Management Policies

Machine learning revolves around the creation and study of computer systems and machines that can learn from data presented to them. Many examples of such systems exist, for example speech recognition systems, spam filters, and robot controlled cars. A formal definition of machine learning was presented by Tom M. Mitchell in his 1997 book on the topic:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with E. [60]

For example, a computer that learns to play a card game might improve its performance with regards to winning that game, as it gains experience by playing it against itself or others. The quality of machine learning systems are dependent on multiple design choices, such as which training data is used, the features extracted from this data, and the chosen algorithm for learning the target function based on the training data.

Machine learning has been used extensively in the context of PM. Examples include powering down idle hard drives and wireless network interfaces [28], [39], [86], putting CPUs to sleep and adjusting core frequencies [54], [9], [61], as well as energy efficient scheduling of tasks [93].

There are many possible ways to train a machine learning system. With *supervised learning*, pre-labeled training data is used to approximate the target function. The key point of supervised learning is that a human, for example some expert in the field of the classification problem, have to correctly label the training data so that it can be used to improve the performance of the algorithm. In contrast, *unsupervised learning* rely on the ability to uncover some hidden structure in raw unlabeled data. *Semi-supervised learning* combine these approaches. Labeled training data is used as an starting point for automatically labeling the remaining data. This has been found to improve learning accuracy considerably in some cases where the generation of a fully labeled training set is unfeasible, while unlabeled data can be obtained cheaply. *Reinforcement learning* is a category of machine learning algorithms that attempt to find ways of maximizing the reward (minimizing the cost) of a system by repeatedly selecting one among a set of predefined actions.

In the context of power management, machine learning approaches are often divided into two classes of systems, namely *pre-trained systems*, and systems employing *online learning*.

Pre-trained Systems

Pre-trained systems commonly employ supervised- or semi-supervised learning to train the employed PM algorithm. Typically, example workloads are used to obtain traces used for training. Unless a representative set of training workloads is used, the algorithm may become unable to generalize when experiencing real loads. Because of the dynamic nature of PM problems, pre-trained systems are less common than online ones. Nonetheless, pre-trained systems have been used successfully for PM, for instance to change CPU frequencies [9], [61], [47].

Online Learning

Online learning encompasses machine learning methods in which no set of separate training samples is used. Instead, the algorithm tries to produce the best prediction possible as examples arrive, starting with the first example that it receives. After making a prediction, the algorithm is told whether or not this prediction was correct, and uses this information to improve its prediction-

hypothesis. Each example-prediction pair is called a *trial*. The algorithm will continue to produce predictions as long as new examples arrive, and the learning and improvement of the hypothesis is thus continuous. Because the learning process never stops, it is sensible to employ an algorithm that is capable of computing successive prediction-hypotheses incrementally, and avoiding the potentially excessive costs of recalculating every hypothesis from scratch (thus having to store the examples indefinitely) [56], [57].

Online statistical machine learning has been used to control CPU frequency [45], while online reinforcement learning has been employed in the domain of energy efficient task scheduling and for managing wireless local area network (WLAN) interfaces [93], [86]. Online machine learning has also been used to spin down hard drives in order to conserve energy [39].

2.4 Power Management Mechanisms

PM can be performed on a wealth of devices. Notable examples are powering down network cards, spinning down hard drives, adjusting fan-speeds, etc. To enable software—for example an OS—to take advantage of available PM capabilities of the hardware, some mechanisms are necessary. First, it must be possible to detect the presence of any PM capable hardware. Second, the configuration of devices (and possibly the software itself) must be performed. Third, the software must have some method to communicate PM decisions to the hardware. The following sections gives a brief overview of different PM mechanisms. Thorough descriptions of these mechanisms are deferred to chapter 4.

2.4.1 Device Detection and Configuration

With myriads of different hardware vendors and rapid technological development, the existence of multiple standards for detection and configuration of devices is not surprising. Examples are the use of vendor- and device-specific BIOS code, and advanced power management (APM) APIs and description tables. Advanced Configuration and Power Interface (ACPI) evolved from the need to combine and refine existing power management techniques such as those mentioned above, into a robust and well defined specification for device configuration and power management. Using this interface, an OS can manage any ACPI-compliant hardware in a simple and efficient manner.

2.4.2 Power Management of Devices

Through the use of for example ACPI, software can detect and configure PM capable devices in the system. ACPI provides information about devices through objects. While the objects normally contain tables of information, they can also contain methods that can be used to communicate software capabilities and initiate state transitions. Consider a PM-capable rotating hard drive drive: through ACPI it presents tables listing its power/performance states. Such tables would likely include the power consumption, bandwidth, and expected read/write latency for each state. Probably, it would also contain how long it takes to enter the different states, and their associated spin-up time—the time spent spinning the disk back up when leaving them. Now, using the information obtained from such tables, and ACPI accessible *methods* for the device, software can decide on what state the disk should reside in at any given time, and then make it enter the chosen state. Figure A.3 in appendix A illustrates the concepts described here, but for entry of CPU performance states.

Although ACPI provides a well defined and robust way of interacting with devices, not all devices support this specification. Legacy devices, and also relatively new hardware, can lack support for ACPI[1]. Sometimes, such devices provide PM capabilities by means of some other specification or technology. Common examples are devices connected via PCI and PCIe interfaces, which define their own methods for performing PM. In addition, some specialized devices have functionality and properties not easily representable through ACPI. For instance, Intel employ both ACPI and their own custom mechanisms for detection, configuration, and power- and performance management of their CPUs [25], [26].

2.4.3 Power Management of CPUs

ACPI provides several mechanisms for controlling the power consumption and performance of CPUs. The two most commonly used (and the ones employed in this work) are power states (C-states), and performance states (P-states). C-states correspond to different CPU sleep-states, that is, states when no instructions can be executed and energy is conserved by powering down parts of the processor. P-states corresponds to combinations of CPU frequency and voltage, higher frequencies results in increased performance but also consume more power.

1. On our hardware, the Gbit NICs are PM capable but via PCI instead of ACPI. In fact, our entire platform is severely lacking in terms of ACPI support.

Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) enables software to dynamically adjust both the CPU voltage and frequency. Typical use of DVS can result in substantial power savings due to the dependency between CPU voltage, frequency, and power dissipation [40]. Consider, if implementing a chip using static CMOS gates, its switching power dissipation is given by equation 2.5, where P is the power, C is the capacitance being switched per clock cycle, V is the voltage, and f is the switching frequency.

$$P = CV^2 f \tag{2.5}$$

As can be seen from equation 2.5, power dissipation is quadratically proportional to the CPU voltage[2]. Because of this, the frequency of the CPU must be reduced when lowering the operating voltage in order to save power. As such, DVS trades performance for energy reductions. In modern CPUs, a set of *frequency/voltage pairs* are often exposed through technologies such as Enhanced Intel® SpeedStep® [27], or AMDs Cool 'n' Quiet™/PowerNow!™[20], [21]. These technologies simplify power management, and ensure that the paired frequencies and voltages are compatible. For this reason, DVS is often also called dynamic voltage and frequency scaling.

2. For a relatively non-theoretical explanation as to why is this is the case, the following blog posts can be recommended:
http://software.intel.com/en-us/blogs/2009/06/29/why-p-scales-as-cv2f-is-so-obvious
http://software.intel.com/en-us/blogs/2009/08/25/why-p-scales-as-cv2f-is-so-obvious-pt-2-2

3

ROPE Architecture and Design

This chapter covers the architecture and design of ROPE, which contains all PM functionality in Vortex. First, the architectural foundations and basic ideas are established. Following this, designs for various PM policies are introduced and discussed in more detail.

3.1 ROPE Power Management Architecture

Power management (PM) of the CPU is critical for saving power. Despite modern CPUs having become considerably more energy efficient in recent years [12], the CPU is still responsible for 25%–60% of the total power consumption of computers [59], [88], [12]. To reduce power consumption, the CPU must be managed with regards to both power- and performance states. In broader terms, this means putting the CPU to sleep and adjusting its frequency.

ROPE introduces PM functionality to the Vortex kernel. In ROPE, management of power- and performance states (C- and P-states, respectively) is performed independently of each other. The architecture of ROPE is founded on two central concepts; *policies*: rules describing the use of power- and performance states; and *enforcement*: effectuating the policy with hardware. As

outlined in section 2.1, different policies can vary significantly in their nature, i.e. they can be static or dynamic, pre-configured or based on runtime metrics. Thus, they may rely on different data structures, libraries, instrumentation[1], etc. Further, different policies might share method of enforcement. For instance, a set of timeout-based policies, each with a distinct timeout value, can rely on the same method for enforcing the timeout, while a more complex policy might need a completely different enforcement.

The design goals for ROPE include simplicity and flexibility. Guided by these, we base the architecture and design of our solutions on the basic ideas of policies and enforcers. An OS kernel is a delicate and complex piece of software, and our solutions are influenced by this environment. The necessary logic, instrumentation, and data structures, must often be incorporated into the Vortex kernel in a scattered and fragmented fashion, leveraging existing code paths and infrastructure. Based on these observations, we deem it appropriate *not* to establish any kind of rigid framework for the implementation of our PM functionality, as such a framework may quickly become an impediment with regards to flexibility. Figure 3.1 outlines the described architecture.

3.2 Design

This section describes several different designs of PM policies. Some of these provide alternative solutions to the same problem, while other policies have different objectives. Policies with different objectives are intended to complement each other, augmenting the Vortex PM system as a whole. Although all CPU PM features will be somewhat intertwined, we propose policies along three orthogonal axes:

- **CPU power state management:** These policies control when to turn a CPU-core off, and what to do when it is not executing instructions, e.g. which CPU C-state to enter.

- **CPU performance state management:** These are policies governing the performance of the CPU core when it is powered on. Typically, this means selecting one of the available CPU P-states.

- **Energy efficient scheduling:** When the individual CPU cores are powered down and clocked according to the workload, it becomes interesting to avoid unnecessary wake-ups. Using energy efficient scheduling,

1. For example hardware instrumentation using performance monitoring counters (PMCs), or software states.

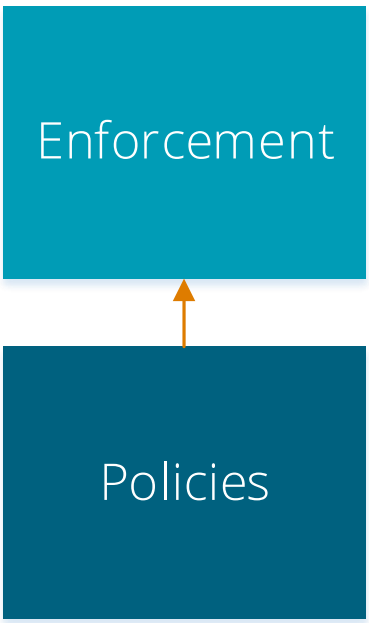


Figure 3.1: Architecture of PM functionality implemented in Vortex.

it is possible to assign tasks to cores such that as many cores as possible are kept inactive, and thus sleeping, at any given time.

A single PM policy is realized through an enforcement mechanism (EM), and the structures and logic necessary to support this EM. These auxiliary structures and pieces of logic provide the EM with necessary information when effectuating PM decisions. Examples of such information could be how long the CPU must remain idle before entering a given C-state, which C-state to enter, and so on. We treat this logic, its data structures, and the information it contains, as a black box, and call it a Policy State Manager (PSM). As long as the PSM is paired with its matching EM, the policy implemented by the two will be effectuated, and the rest of the Vortex kernel will remain entirely independent from the introduced PM policy.

Modern CPUs commonly provide the possibility of adjusting the voltage and frequency of sub-components, or voltage/frequency islands (VFIs) within a single chip. For example, recent AMD Opteron processors support independent frequency control of all four cores, the shared L3 cache and on-chip northbridge, the DDR interface, and four HyperTransport links [46], [40]. In accordance with this trend, all PM functionality implemented in ROPE oper-

ates at a per-core level. This design choice results in a high degree of flexibility, which is especially desirable in the context of a VMM. For example, in the current version of the Vortex kernel all interrupts related to network traffic are handled by a single CPU core[2]. For some application scenarios, it might be necessary to ensure maximum performance with regards to network latency. With the per-core PM approach, a less aggressive policy[3] can be configured for the core handling the network traffic. Further, different tenants can be configured to run on separate cores, which in turn are configured to use a PM policy matching the customer's desired performance/carbon footprint.

The complexity of different PSMs will vary with the policies they support. In the following paragraphs, different PM policies available in Vortex are described. Some of these are relatively straight-forward and have very simple PSMs, while others are more sophisticated.

3.2.1 CPU Core Power State Management

ROPE supports three different policies for managing CPU C-states. This section covers their design.

Aggressive Entry of C-states

The simplest PM policy in Vortex is to enter a given C-state aggressively whenever a core is idle. With this policy, the enforcer simply needs to know *which* C-state to enter, and then execute a state transition. The design for this policy is illustrated in figure 3.2.

Static Timeout based C-state Entry

With static timeout-based policies, a constant timeout value is used to determine whether or not a C-state should be entered. The device being managed has to remain inactive throughout the duration of the timeout for the low power state to be entered. In this policy, the PSM is responsible for providing the static timeout value to the enforcement module. Figure 3.3 illustrates the

2. Using only a single core for handling network interrupts is a configuration choice. On the current hardware, it is most performance inducive that the same core both insert packets into the transmit queue, and remove packets from the receive queue. This is because both of these queues are protected by the same lock, and using a single core thus avoids lock contention. Experiments have shown a 5.5% increase in CPU consumption if more than one core is used [52].
3. PM could even be deactivated for this core all together.

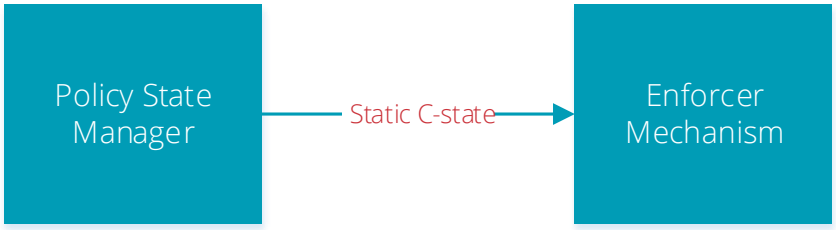


Figure 3.2: Design of a policy that aggressively enters C-states.

overall design of this solution.

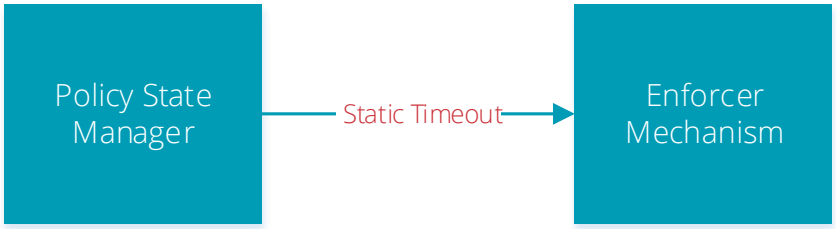


Figure 3.3: Design of a policy for entering a C-state following a static timeout.

Select Best Performing C-state

This policy employs an algorithm that dynamically selects which C-state to enter at runtime. This is done at the granularity of individual CPU-cores. The PSM for this policy is composed of two main sub-components, each serving a distinct purpose:

- **Power state logger:** Performs runtime logging of the current power state of a CPU-core.
- **C-state selection module:** Calculates the power saved and latency added as a result of aggressively entering different C-states. Based on these calculations, the currently best performing C-state is chosen.

Figure 3.4 illustrates the design of the policy where the best performing C-state is selected.

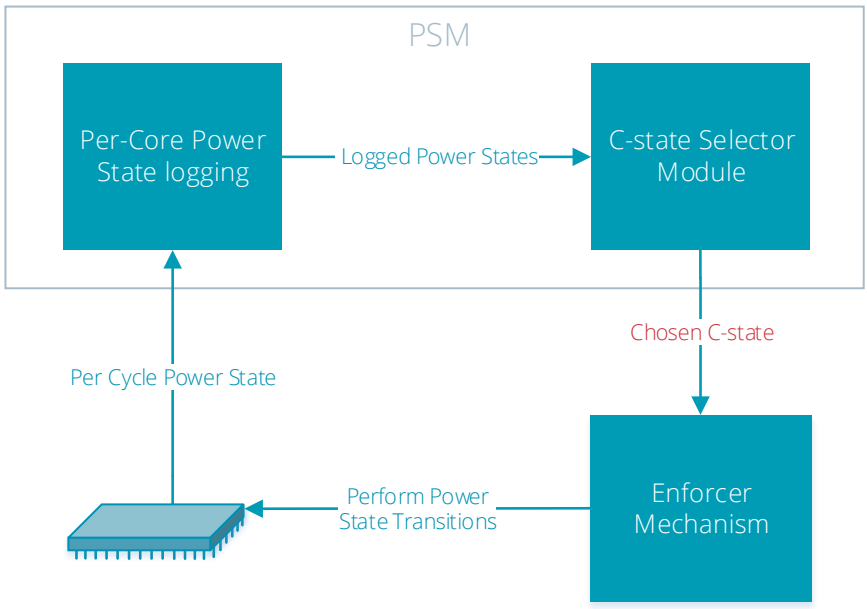


Figure 3.4: The design of a policy where the best performing C-state is chosen.

3.2.2 CPU Core Performance State Management

While the previous section outlined different policies for conserving energy using CPU C-states, this section considers the design of a policies for controlling *performance* states on a per-core basis. These policies and their components are responsible for adjusting the operating frequency of an individual core according to the workload. We employ two workload prediction techniques for selecting which P-state to use at any given time, both of which share the same general design.

The PSM for supporting dynamic selection of P-states consists of two key components: a *Performance State Logger* and a *Prediction Module*. Figure 3.5 illustrates the design of our P-state management solution.

- **Performance state logger:** Performs runtime logging of the current performance state of a CPU-core.
- **Prediction module:** Generates predictions about which P-state to use. Relies on records of previously experienced workload patterns in order to make these predictions.

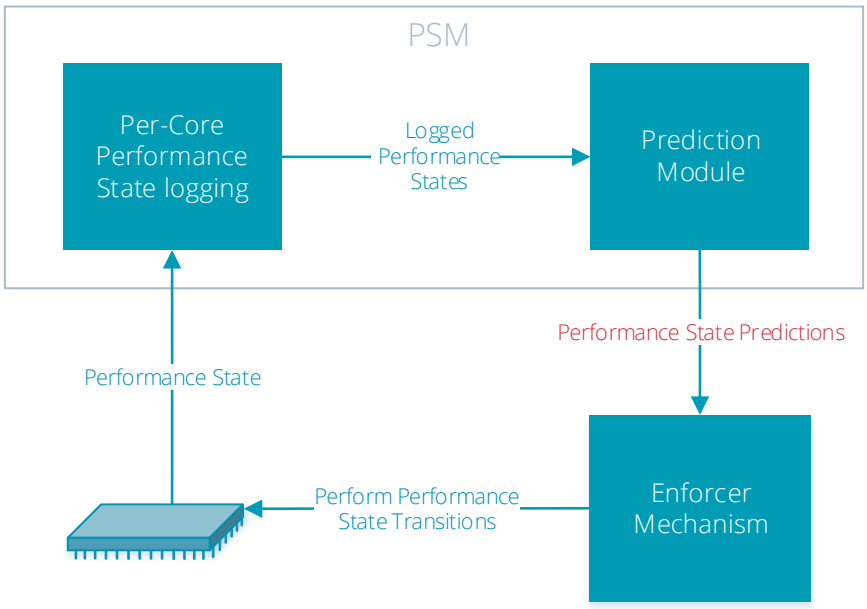


Figure 3.5: The design of a policy capable of managing CPU performance states.

3.2.3 Energy Efficient Scheduling

The third and final branch of PM algorithms implemented in ROPE is that of energy efficient scheduling. In essence, this problems boils down to keeping the maximum number of cores idle for as long as possible. There are several issues that must be considered when designing an energy efficient scheduling solution:

- **Multiple cores per chip:** Modern CPUs often have multiple cores per chip, or *package*. Although some CPUs, such as the previously mentioned AMD Opteron, feature voltage/frequency islands (VFIs) per core, many do not. On our platform, all cores on a package share the circuitry providing the voltage. This means that the voltage must be maintained at a level corresponding to the needs of the most voltage-hungry core on the chip. If the maximum amount of power is to be saved, the CPU-topology must be taken into account. For example, under light load, maximum power savings can be achieved when all the cores in a physical package are completely loaded before distributing the load to another idle package [83].

- **Cache warmness:** To keep caches warm, it is undesirable to unneces-

sarily move threads between CPU-cores. This results in an interesting tradeoff: distribute the total load evenly to all active cores, making sure that the lowest possible voltage can be drawn; or optimize for cache warmness and performance, risking that a single core increases the voltage requirements of the entire package.

- **Cache Contention:** Although maximum power savings can be achieved by loading a single package as much as possible before recruiting an inactive one, this can lead to suboptimal performance. The reason for this is that cores on the same package often share lower level caches. If many threads are scheduled on the same package, contention for these shared resources will occur. The performance impact of this will be highly dependent on the behavior of the scheduled tasks: if the tasks are not memory or cache intensive, performance impact will be minimal [83].

We do not wish to break our design principle of orthogonality, and have our energy efficient scheduler directly alter the C- or P-states of the different cores. This would certainly complicate the coexistence with existing policies managing CPU-states. Instead, we opt for an design where the scheduler is decoupled from the hardware, and attempts to save power while working with abstract *proxies* of the physical CPU-cores. A *Core Selector* implements a scheduling policy while interacting with a *Proxy Manager*. The Proxy Manager holds the state of the physical CPU-cores, such as their current utilization, whether they are active or sleeping, and so on. Together, these components constitute the PSM. This PSM is hooked directly into the existing scheduling framework of Vortex, which serves as the EM. This design is illustrated in figure 3.6.

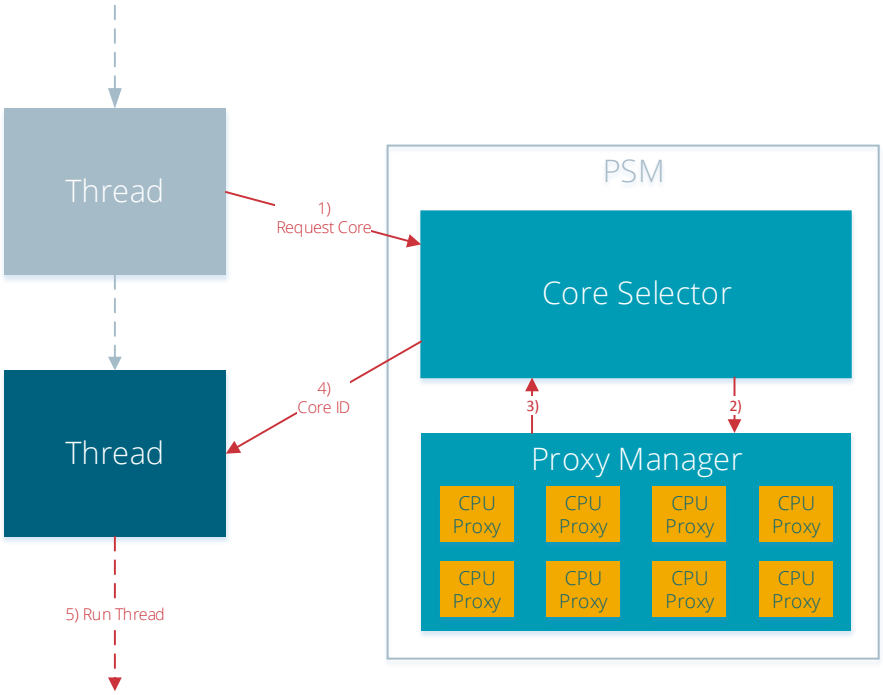


Figure 3.6: Design of an energy efficient scheduler.

4

Implementation

In this chapter, we describe the implementation of several PM policies along the three orthogonal axes of C-states, P-states, and energy efficient scheduling. First, we describe some of the mechanisms that are used in ROPE to control the hardware. We then explain a online machine learning algorithm used in one of our P-state predictors. Lastly, we describe the implementation of each of the PM policies available in ROPE, and provide insights into the costs of operating these.

4.1 ACPI

ACPI is a method of describing hardware interfaces in a terms abstract enough to allow flexible hardware implementations, while being concrete enough to allow standardized software to interact with such hardware interfaces [22]. The ACPI specification was created to establish common interfaces that enable robust OS-directed motherboard device configuration, and power management of devices and systems. ACPI is a fundamental component for Operating System-directed Power Management (OSPM).

ACPI evolved from the need to combine and refine existing power management BIOS code, advanced power management (APM) APIs, and description tables, etc., into a robust and well-defined specification for device configuration and power management. Using an ACPI-specified interface, an OS can

manage any ACPI-compliant hardware in a simple and efficient manner.

At the heart of the ACPI specification are the ACPI System Description Tables. These describe a specific platform's hardware, and the role of ACPI firmware is primarily to supply these ACPI tables (rather than a native instruction API) to PM software [22]. The System Description Tables describe the interfaces to the hardware, and some of these tables include "definition blocks" that contain ACPI Machine Language (AML): an interpreted, and abstract machine language. Using this language, hardware vendors can describe their hardware—certain that any ACPI-capable OS can interpret it correctly. The ACPI System Firmware, in addition to supplying description tables, also implements interfaces that can be used by the OS. For example, if the description tables contain an ACPI-object for restarting a device, this object (which the OS can interact with) will be backed by firmware that controls the actual hardware.

Figure 4.1 shows the architecture of ACPI. ACPI itself (shown in red) is independent of both the underlying hardware and the OS. Any OS that wishes to support ACPI must provide drivers, and a AML parser- and interpreter. If these components are in place, any interfaces and information presented by the platform can be accessed and used. Similarly, the hardware is not a part of ACPI per se. However, if the hardware wishes to expose functionality to the OS, it can do so through ACPI-objects accessible through the ACPI system description tables.

4.1.1 Supporting ACPI in Vortex

The ACPI standard consists of multiple components that allow the setup and power management of various devices. These components include ACPI namespace management, ACPI table and device support, ACPI event handling capabilities, and AML parser/interpreter functionality. The ACPI Component Architecture (ACPICA), developed by Intel Corporation, defines and implements these software components in an open-source package. One of the major goals of this architecture is to isolate the operating system from the code necessary to support ACPI. Most of the ACPICA code base is independent of the host OS in which it exists, using only a thin translation layer to access necessary OS functionality. To port ACPICA to a new operating system, only this OS specific layer must be implemented. This is necessary because every operating system is different, and the way an OS provides necessary services such as synchronization, input/output, or memory allocation, can differ enormously [24].

The ACPICA subsystem resides within the OS, and all interaction between

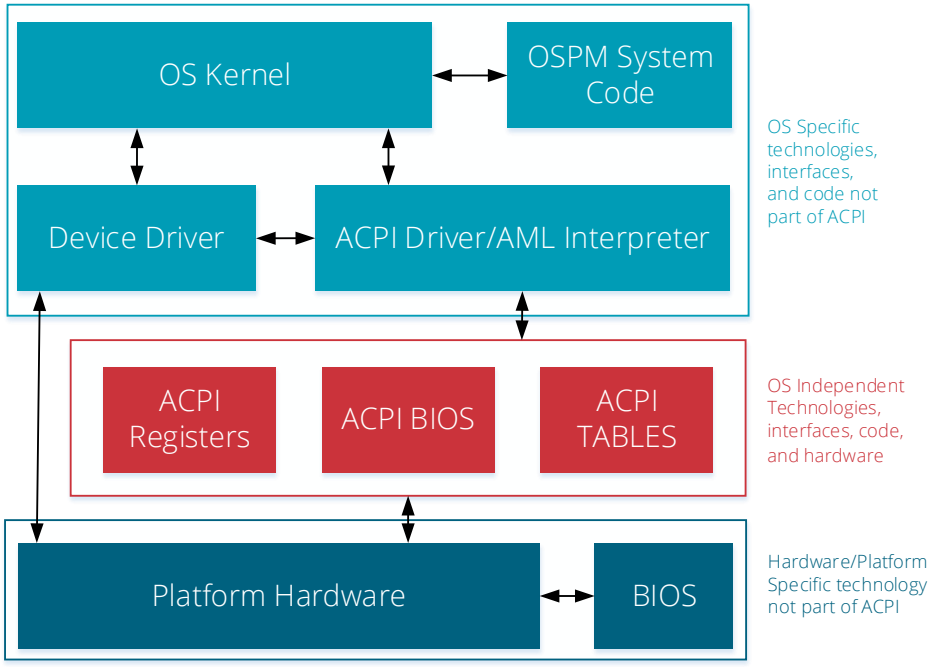


Figure 4.1: Architecture of ACPI.

the host operating system and the OS-independent ACPICA components is performed through the thin service layer. This is illustrated in figure 4.2. In our case, the OS Service Layer (OSL) is implemented with approximately 1000 lines of Vortex-specific C code.

The OSL consists of two interfaces. These interfaces are used to support access to the host OS functionality as explained above, but also allow the host OS access to the functionality of the ACPICA subsystem, for example setup and power management of devices. A sketch of this can be seen in figure 4.3.

4.1.2 CPU Performance and Power Management using ACPI

ACPI provides three mechanisms for controlling the performance states of processors. These are:

- Processor power states, named C0, C1, C2, ..., C_n.
- Processor clock throttling.

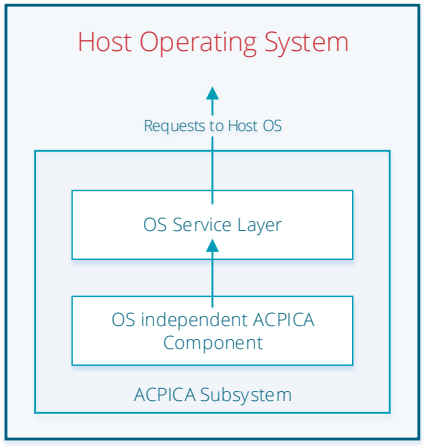


Figure 4.2: The figure illustrates how the ACPICA-subsystem resides within and interacts with the host operating system.

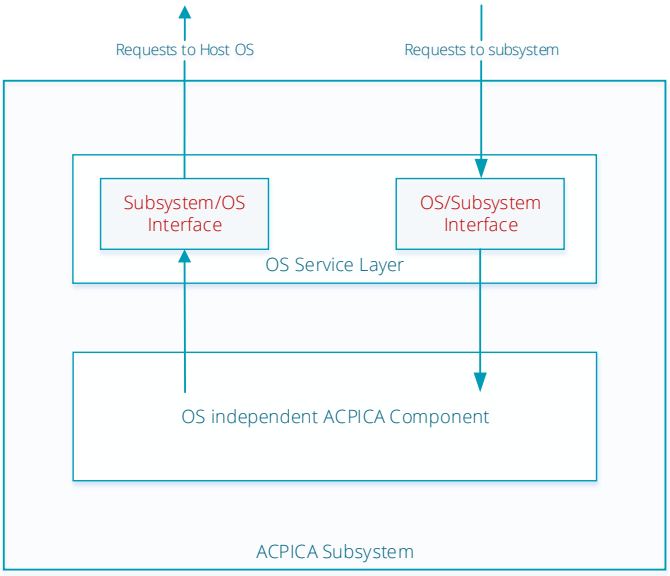


Figure 4.3: The figure illustrates how the operating system service layer of the ACPICA subsystem is structured into two interfaces, together facilitating the necessary bindings between the subsystem and the host OS.

- Processor performance states, named P0, P1, P2, ..., P_n.

If supported by the platform, the OS can use these mechanisms to save power while attempting to meet performance constraints. Central to managing both performance and power are the *processor power states*. These states, denoted C0 through C_n, are used conserve power by halting the execution and shutting down internal processor components. The C0 state is the only active state (where execution occurs), and all other power states (from C1 up to Cn) are sleep states. While in a sleep state, the processor is unable to execute instructions. Higher-numbered sleep states conserve more energy. Each sleep state also has an associated entry and exit latency, which also typically increases with energy savings. All processors that meet ACPI standards *must* support the C1 state (which corresponds to the HLT instruction), whereas all other processor power states are optional. Further, the C1 (and C2 if supported) state rely on the platform maintaining state and keeping caches coherent. In addition, the latency of the C1 state must be kept sufficiently low that "OSPM does not consider the latency aspect of the state when deciding whether to use it" [22]. Aside from putting the processor in a lower power state, entering C1 should have no software-visible consequences. The same goes for the optional C2 state, differing only from the C1 state by its higher latency and greater energy savings. The even deeper sleep states—C3 and beyond—rely on the OS to maintain cache coherency, for example by flushing and invalidating the caches before entering the C3 state. When in the C0 power state, the use of *throttling* and *processor performance states* enables fine grained control of performance and energy expenditure.

The relationship between processor power- and performance states, as well as clock throttling is illustrated in figure 4.4. While in the C0 state, CPU P-states and clock throttling can be used to adjust the performance level. From C0, C1–C_n can be entered using entry methods defined in ACPI objects. Note that the C1 state is available through the HLT instruction. When in a sleep state, C0 can again be entered if an interrupt occurs, or for higher-numbered C-states, in the occurrence of a busmaster access.

Processor performance states are—as their name implies—used to adjust the performance of a processing unit. As they all allow execution of instructions, they reside within the C0 power state. The performance states are named P0 through P_n, where P0 is the state offering the highest level of performance. As with C-states, a variable latency is associated with entering and exiting a P-state. P-states offer OSPM the opportunity to alter the operating frequency for the processor. For instance, the Intel Xeon x5355 processors offers three P-states (P0, P1, and P2), at 2.66, 2.33, and 1.99GHz respectively.

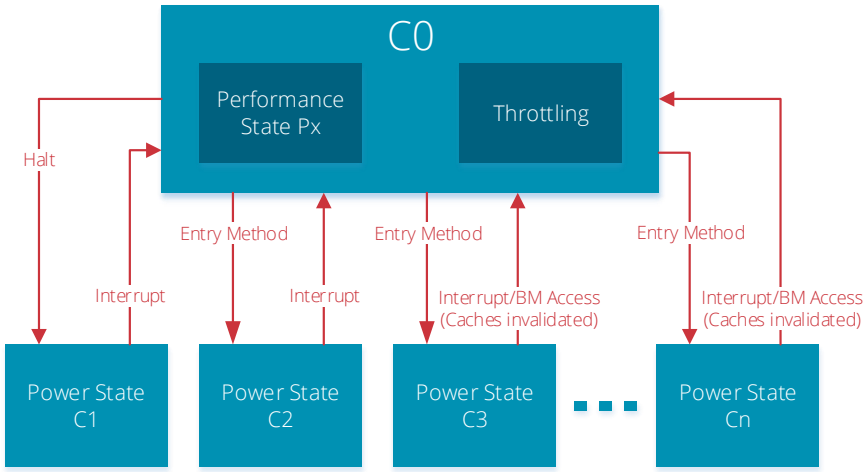


Figure 4.4: The figure illustrates how the the processor power- and performance states, as well as throttling capabilities are organized. This figure is loosely based on figure 8-39 of the ACPI specification version 5.0.

To facilitate the above mentioned functionality, a myriad of different ACPI objects must be presented by the platform and used by the software implementing OSPM. Appendix A gives a brief introduction to some of the most essential of these objects.

4.2 Intel Specific Advanced Power Management

ACPI provides a general framework and specification for power management and device configuration. Alas, this generality comes at the cost of reduced flexibility and lack of the precision necessary to efficiently manage the available PM features of modern CPUs. On Intel platforms, these shortcomings are remedied by the introduction of vendor specific instructions. The use of such instructions have enabled considerable power savings in mature OSs. One recent example of this is the shift away from the traditionally ACPI-based CPU governors, towards vendor-specific CPU drivers within the Linux kernel [7]. Similarly, AMD-specific patches have been added to Linux CPU governors to increase power-savings[1]. In the following sections, a brief introduction will be given to the Intel-specific instructions used in the implementation of ROPE.

1. http://comments.gmane.org/gmane.linux.kernel.cpufreq/9938

4.2.1 The CPUID Instruction

The cpu Identification (CPUID) instruction returns processor identification information. It is important w.r.t. PM as it provides information about what PM features are available for a given processor. Sometimes, information retrieved via the CPUID instruction can be obtained through ACPI tables as well, but this is not always the case. Further, information obtained via CPUID and ACPI might be similarly named, but their values might differ[2]. Examples of PM-relevant information available through CPUID are listed below:

- Processor-specific information regarding the MONITOR/MWAIT instruction pair, e.g. the number of available C-states and the size of monitor lines.

- Information about architectural performance monitoring, for example the number of available general purpose performance monitoring registers, and availability of different performance monitoring events[3].

- Information regarding the platform topology, for example which logical processors share the same cache.

Because it conveys a lot of information, the CPUID instruction is valuable when attempting to save energy on platforms employing Intel CPUs. In the following section, the MONITOR/MWAIT instruction pair will be discussed in more detail.

4.2.2 The MONITOR/MWAIT Instruction Pair

MWAIT relies on the use of hints to allow the processor to enter implementation-specific optimized states. There are two usages for the MWAIT instruction pair, namely *address-range monitoring* and *advanced power management*. Either of these rely on the use of the MONITOR instruction [25].

The MONITOR instruction allows the CPU to wait while monitoring a given address-range. If the necessary extensions are provided by the platform, the CPU can enter different C-states by passing *hints* to MWAIT. The availability of MWAIT extensions for PM can be verified through CPUID [26]. In addition to being used for PM, the monitor instruction can also be used for other purposes,

2. On our platform, the ACPI tables list only the C1 and C3 power states, while CPUID provides information about six states (C1 through C3, each with two sub-states).
3. The number of LLC misses used for regulating CPU P-states is one example of such events.

for instance spinlocks[4]. In fact, thread synchronization was MWAIT's original purpose [6].

According to the ACPI specification, the C1 power state must always be available. This power state can be entered either by executing the HLT instruction or, where available, by MONITOR/MWAIT with the correct hint. However, using the HLT instruction in Vortex is problematic. As will be explained in section 4.4.2, interrupts must be enabled in the idle function. Because of this, an interrupt may alter the idleflag at any given time, for example, directly following it being checked to verify whether the HLT instruction should be issued. If this happens, the result will be that the core receives work (i.e. it is interrupted), but instead of breaking out of the idle function, the CPU-core is halted. This is disastrous as no new interrupt will arrive[5]. Rather, we use MONITOR/MWAIT with hints to enter all C-states, including C1.

Code listing 4.1 shows the function implementing entry of C-states using the MONITOR/MWAIT instruction pair. While looping on the boolean value of the CPU-core's idleflag, a monitor is armed on the address of this flag. Next, MWAIT is performed on the address. The second hint argument determines which C-state is entered.

The loop is necessary because of a phenomenon called *false wakeups*. MONITOR/MWAIT works by monitoring a given memory location. However, it may also trigger whenever a cache line containing this address is evicted, even when the monitored address remains unchanged. To avoid such false wakeups, it is common to add padding around variables that are to be monitored. Recall that the size of the monitor lines (and thus the size of the pads) can be retrieved using the CPUID instruction.

4.3 The Share Algorithm

In this section we introduce the machine learning approach that we use to select the best among a population of C-states. The algorithm belongs to the *multiplicative weight* algorithmic family, which has proven itself to provide good performance for several different on-line problems [35], [56], [57]. Algorithms in this family are all based on the same basic premise; a *master*

4. https://blogs.oracle.com/dave/resource/mwait-blog-final.txt
5. It is possible to work around this. One solution is to inspect the previous instruction pointer (on the stack) when entering the interrupt handler. If the interrupted instruction lies within the code region performing the testing of the idleflag and execution of the HLT, the interrupt handler could advance the instruction pointer such that the HLT instruction is bypassed when returning from the interrupt handler.

```
1  inline void mwait_idle_with_hints(uint32 extensions, uint32 hints)
2  {
3      while (IDLEFLAG == TRUE ) {
4          if (IDLEFLAG == TRUE) {
5              asm_monitor((vaddr_t)&cpulocal[CPU_ID].cl_idleflag, NEXTENSION, NHINT);
6              if (IDLEFLAG == TRUE)
7                  asm_mwait(extensions, hints);
8          }
9      }
10 }
```

Listing 4.1: Usage of MONITOR/MWAIT instruction pair.

algorithm relies on a set of subordinate algorithms, *experts*, for input. Every trial, each of these experts make predictions, and the master's goal is to combine these predictions in the best way possible, minimizing some cost or error over a sequence of observations. The master associates a *weight* with each of the experts, and these weights represent the quality of each of the experts' predictions. Using these weights, the master can select the best performing expert, or produce a weighted average of the set of predictions.

After a trial has been completed, the weights of the experts are updated according to the quality of their individual predictions. If the expert's prediction was incorrect, the weight of the expert is reduced. If the prediction was correct, the weight is not changed. Normally, this method causes the predictions of the master algorithm to converge towards those of the best performing expert. However, this can be problematic. Across a series of sequences, the error of an expert can be arbitrarily large, and because of this, its weight arbitrarily small. This results in the master algorithm being unable to respond quickly to changes.

The Share Algorithm, as defined by Herbster and Warmuth [42], derives its name from a solution to the above mentioned problem. It allows small weights to be recovered quickly, and does so through an additional update step called the *Share Update*. In this step, each expert shares a fraction of its weight with the other experts. This fraction can either be fixed, or variable. In the *Variable Share* algorithm, the fraction of weight that is shared is proportional to the error of the expert in the current trial. If an expert makes no error, it shares no weight.

4.3.1 Definition

Let x_1 to x_n denote the predictions of n experts. The weights of the experts, denoted ω_1 to ω_n, are initially set to $1/n$. For each trial, we use the notation

$Loss(x_i)$ to refer to the loss, or error, of expert i. The share algorithm takes two additional arguments: the *learning rate*, $\eta > 1$, which controls the rate at which the weights of poorly performing experts are reduced, and the *share parameter*, $0 < \alpha < 1$, which controls at which rate a poorly performing expert is able to recover its weight when it starts performing well.

We use the Share Algorithm to select the best out of a set of experts, saving energy by powering down CPU-cores. We borrow notation from [54], but our definition contains a loss function tailored to our use-case. For each trial, the algorithm completes the following steps:

1. Select the expert x_i with the highest weight ω_i.

2. Compute the loss of each expert x_i at time t

$$Loss(x_i, t) = c_{\text{wasting}} E_{\text{wasted}} + c_{\text{latency}} t_{\text{unpredicted}}, \tag{4.1}$$

   where E_{wasted} is the total amount of energy wasted by using only expert x_i compared to the optimal combination of all available experts, and $t_{\text{unpredicted}}$ is the total unpredicted power-up time for the core under the policy of expert x_i. The weighting parameters, c_{wasting} and c_{latency}, are used to quantify the preference for avoiding latency and of being in lower power states.

3. Reduce the weights of experts that are performing poorly

$$\omega_i' = \omega_i e^{-\eta Loss(x_i)} \tag{4.2}$$

4. Share some of the remaining weights

$$pool = \sum_{i=1}^{n} \omega_i' \left(1 - (1-\alpha)^{Loss(x_i)}\right), \tag{4.3}$$

$$\omega_i'' = (1-\alpha)^{Loss(x_i)} \omega_i' + \frac{1}{n} pool \tag{4.4}$$

5. Rescale the weights to avoid underflow due to continually shrinking weights

$$weightsum = \sum_{i=1}^{n} \omega_i'', \tag{4.5}$$

$$\omega_i^{\text{final}} = \frac{\omega_i''}{weightsum} \tag{4.6}$$

The new weights, ω_i^{final}, are used in the next trial. The above stated algorithm runs with time complexity of $O(n)$, where n is the number of experts. Also, the experts could be executed in parallel [57]. Essentially, the bulk of the cost of running the Share Algorithm can be attributed to the subordinate expert algorithms; as long as these are efficient, the Share Algorithm will be efficient.

4.4 Per-core Power State Management

This section covers the implementation of PM policies and Policy State Managers (PSMs) governing the use of CPU power states (C-states). We describe how these policies and their supporting structures are incorporated into the Vortex kernel, and present implementation specific costs.

4.4.1 Aggressive Entry of C-states

Typically, whenever an OS is out of work to perform, some kind of idle state is entered. In Microsoft Windows, this functionality is implemented through the System Idle Process [78, Ch. 5]. Linux offer similar functionality through what is called *idle*, or *swapper* tasks. These are simply tasks with the lowest possible priority, which means that they only will be scheduled if no other tasks are runnable.

In Vortex, a CPU-core receives work through message queues. When no new messages are present, the core is out of work and goes idle by invoking the *idle()* function. As soon as a new message is presented to the core, it will leave the idle function and start executing the received task.

The aggressive entry of a C-state is a very simple PM policy. In fact, which C-state to enter is simply read from CPU-core local data structures, and this variable forms the PSM in its entirety. The only overhead from this policy is that of reading the C-state, and transitioning in and out of it. The transition into the selected C-state is accomplished through the use of the MONITOR/MWAIT instruction pair (see section 4.2.2). Listing 4.2 shows an excerpt of the idle function modified to aggressively enter a chosen C-state.

4.4.2 Static Timeout Based C-state Entry

In previous work, increasing timeout values for dynamic timeout-based PM has been used to reduce the latency experienced by the user [54], [39] It is

```
1  void idle(void)
2  {
3      /* Initialization */
4              .
5              .
6
7      // Enable interrupts
8      _INTERRUPT_ENABLE_UNCONDITIONAL();
9
10     // Loop till something happens
11     while (IDLEFLAG == TRUE) {
12
13         // Ensure power management components activated
14         if (cpulocal[CPU_ID].cl_pm_active) {
15
16             // Enter C-state
17             mwait_idle_with_hints(NEXTENSIONS, cpulocal[CPU_ID].cl_cstate_hint);
18         }
19     }
20
21     // Disable interrupts
22     _INTERRUPT_DISABLE_UNCONDITIONAL();
23
24     // Dispatch CPUMUX thread
25     arch_thread_dispatch(&cpumux_tcbref[CPU_ID]);
26
27     while (1) ;
28 }
```

Listing 4.2: Idle function accommodating aggressive entry of C-states.

easy to create a scenario where this approach will be correct: whenever the idle function is entered, we busy-wait for the duration of the timeout before entering a deeper C-state. However, this wastes energy. According to the ACPI specification [22], the C1 power state should be cheap enough to enter and leave that the OS can ignore any costs associated with using it. Our platform indicates that the cost is one microsecond.

By using the C1 power state, we can alleviate the energy issue of busy-waiting. Instead of busy-waiting, we can start by entering C1 (which should incur little cost), and then enter the deeper C-state when the timeout expires. This way, we provide low latency close to that of busy-waiting while saving more energy. We can implement this using timers. As soon as the idle function is entered, a timer for the configured C-state is started and the CPU-core halted. If the core is awoken due to the arrival of a message, the C-state timer is aborted. If this do not happen and the C-state timer fires, the corresponding C-state is entered.

Although conceptually simple, the problem is more complex when studied closely. Within the idle function, interrupts *must* be turned on. If interrupts were turned off, incoming work would be unable to trigger the reactivation of the core. As mentioned above, as soon as the idle function is invoked, a timer object will be created and activated. Following this, the CPU-core is halted with the MONITOR/MWAIT instruction pair. At this time, there is only two

possible ways for the core to leave the halted state.

1. Work arrives, and the core is awoken to process the task.

2. No work arrives before the timer fires, and the core enters a sleep state.

In the first case, the timer must be canceled before the core can go on to performing calculations, otherwise it will trigger an interrupt when it fires. To do this, interrupts must be disabled. An interesting situation occurs if the timer fires while the core is in the process of canceling it (interrupts still being disabled). In this case, the timer object is correctly removed, but since the x86_64 platform lacks support for canceling requested interrupts, the interrupt from the timer is queued. When interrupts are finally turned on, this timer interrupt must be handled. In Vortex, such spurious interrupts are simply ignored as no pending timeout (timer object) can be associated with it. As long as these spurious interrupts does not happen very often, the cost in terms of lost CPU-cycles can be ignored. However, when the timeout values become very short, this situation will occur increasingly often. This is somewhat similar, although not as dramatic, as the live lock situation described in [63], where so much time is spent handling interrupts that no useful work can be performed.

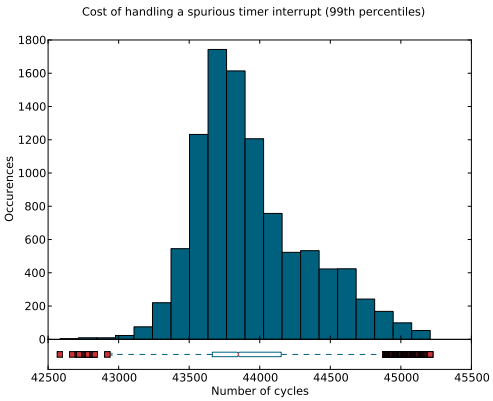


Figure 4.5: The distribution of cycles necessary to handle a spurious timer interrupt. Note that only values within the 99th percentile are included. This is done for the sake of visualization.

The actual cost of handling a spurious interrupt in Vortex is shown in figure 4.5. Note that to avoid over plotting, only measurements within the 99th percentile are included. Further, these figures are based on 10 000 measurements.

In addition to decreasing the opportunity of conserving power, the overhead

of the PM algorithm can affect the user-perceived performance of the system. From the perspective of the user, additional overhead from *entering* the idle state might go unnoticed, since the system is not actively used. However, any overhead when *leaving* the idle loop is critical, as this will add to the user perceived delay. As can be seen in the plots of figure 4.6, the added latency of our solution is low, and. On average, it only adds only 10^3 to 10^4 cycles, or about $0.5\mu s$–$5\mu s$ if the CPU is run at 2GHz.

Code listing 4.3 shows an excerpt from the idle function modified to support the static timeout based policy. Error handling and initialization code have been left out for the sake of brevity.

4.4.3 Select Best Performing C-state

We use the Share Algorithm, as explained in section 4.3, to select the best performing C-state at runtime. Although the algorithm is computationally efficient and simple to implement, running it in a OS-kernel complicates matters. For example, floating point arithmetic can only be performed in the context of a thread[6]. Further, since the code is run frequently, and in a time sensitive environment, we have applied optimizations to reduce both the memory footprint and CPU-consumption. The implementation of the PM policy and PSM for the selection of the best performing C-state is illustrated in figure 4.7. A thread updates the experts' weights and selects the one performing the best for use. This information is then stored in a per-core data structure, and read from the idle function whenever an idle period occurs. Within the idle function, C-states are entered and a residency-log is maintained. This log is communicated to the thread which selects the best expert periodically. Together, these components form a continuous feedback loop.

Measuring CPU-idleness

As mentioned in section 4.4.1, whenever a CPU-core is out of runnable tasks, it enters the idle function. As a consequence, this function is ideal for instrumenting and logging the activity of CPU-cores. By logging timestamps indicating when CPU power states have been entered or left, a high-fidelity trace of the CPU activity can be generated. This approach prevents one critical issue that can occur if for instance statistical sampling is used, and in addition, avoids the extra overhead of having to run a thread or process performing this sampling.

6. Why this is, is covered later in this section.

```
1  void idle(void)
2  {
3      /* Initialization */
4          .
5          .
6
7      // Enable interrupts
8      _INTERRUPT_ENABLE_UNCONDITIONAL();
9
10     // Loop till something happens
11     while (IDLEFLAG == TRUE) {
12
13         // Ensure power management components activated
14         if (cpulocal[CPU_ID].cl_pm_active) {
15
16             //Post a C3 ready timer with a static timeout
17             vxerr = timer_post(&cpulocal[CPU_ID].cl_C3_timerref, cpulocal[CPU_ID].cl_C3_timeout, set_C3_ready_flag, "",
                       NULL);
18
19             // Now enter C1, awaiting work arriving or timer to fire
20             if (IDLEFLAG == TRUE) {
21                 mwait_halt(NEXTENSIONS, C1_HINT);
22             }
23
24             // When returning from the halt, this can either be because of...
25             //  1) The C3 timer having fired -> enter C3.
26             //  2) Because some other thread is ready for work -> abort the timer
27             if (cpulocal[CPU_ID].cl_C3_ready && (IDLEFLAG == TRUE) ) {
28                 cpulocal[CPU_ID].cl_C3_ready = (bool_t)FALSE;
29                 c3_entered = arch_timestamp_usec();
30
31                 mwait_idle_with_hints(NEXTENSIONS, C3_HINT);
32             } else {
33
34                 // Abort the C3 timer
35                 // If the timer has fired in between our check and now, set the C3 ready flag to false
36                 if (!VxO_REFLOCK(&timer_lock, &cpulocal[CPU_ID].cl_C3_timerref))
37                     cpulocal[CPU_ID].cl_C3_ready = (bool_t)FALSE;
38                 else {
39                     vxerr = timer_remove(&cpulocal[CPU_ID].cl_C3_timerref);
40                 }
41                 VxO_XUNLOCK(&timer_lock);
42             }
43         }
44         }
45
46     // Disable interrupts
47     _INTERRUPT_DISABLE_UNCONDITIONAL();
48
49     if (DEBUG_CPUMUX_IDLE)
50         syslog(LOG_DEBUG, "Idle on core %d resuming cpumux", CPU_ID);
51
52     // Dispatch CPUMUX thread
53     arch_thread_dispatch(&cpumux_tcbref[CPU_ID]);
54
55     while (1) ;
56 }
```

Listing 4.3: Idle function with code for entering C-states after static timeout.

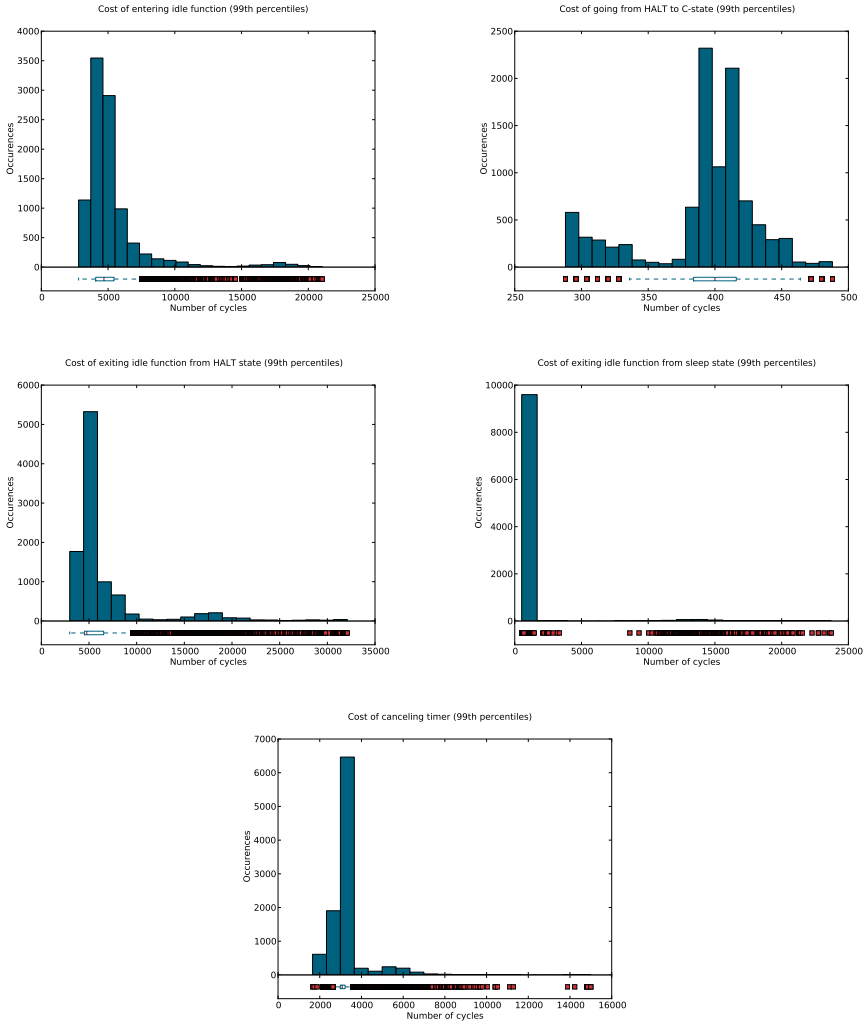


Figure 4.6: The above plots displays added overheads within the idle function originating from the static timeout based policies. Note that for the sake of visualization, only values within the 99th percentile are included. Note that all the graphs are based on 100 000 measurements.

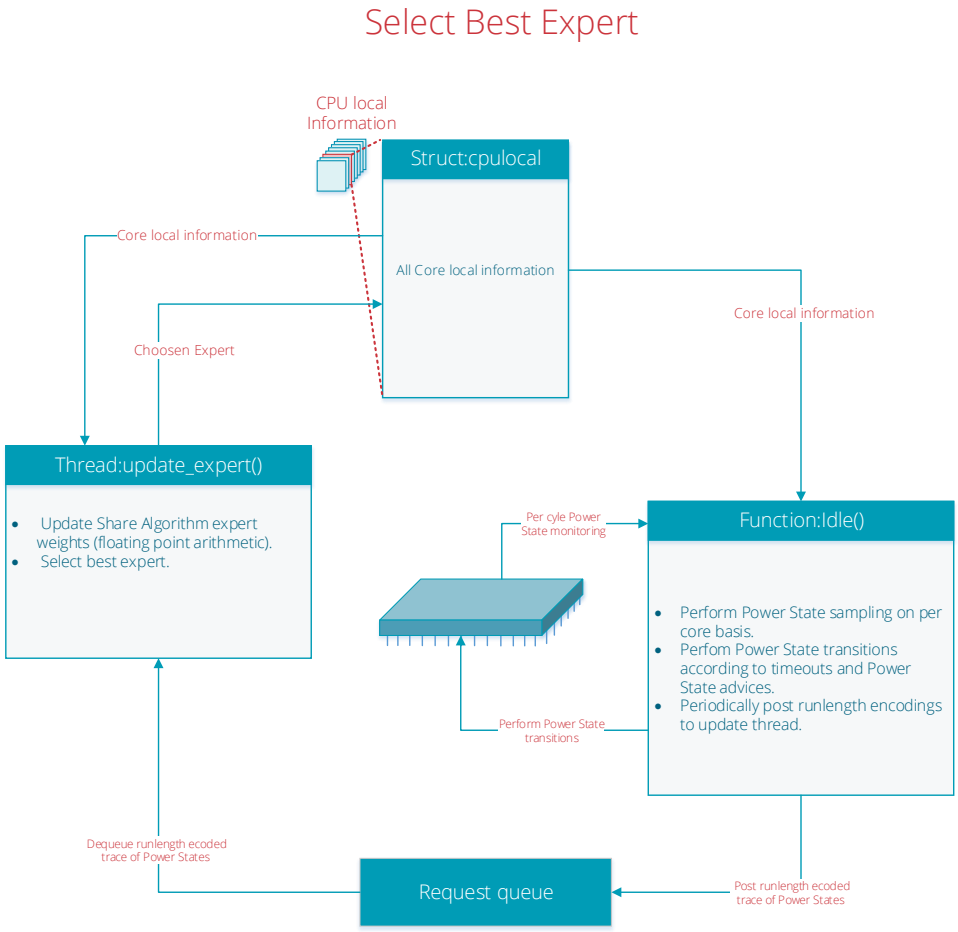


Figure 4.7: The implementation of the policy selecting the best performing C-state.

One of the problems with sampling, is being sure that samples are collected frequently enough. If a work item issued to a processor is sufficiently small, the core can finish executing the task within the time-window of a single sample. This situation is illustrated in figure 4.8. In essence, this means that the trace generated from the sampling process will be incomplete, and no record of the execution of the task exists. This, in turn, might lead to significant errors when evaluating the efficiency of different power saving algorithms working on such traces—in effect rendering the experts weights meaningless.

Instead, we log timestamps of entry and exit of C-states in the idle function itself. This way, we are able to maintain a high fidelity log of time spent by the core in all C-states.

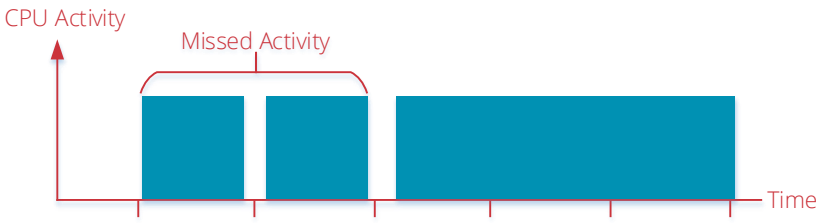


Figure 4.8: The figure illustrates the problems that can occur if sampling is used to measure CPU activity.

Run Length Encoding

To reduce the memory overhead of the above mentioned log keeping, we employ a technique called run-length encoding (RLE). This is a lossless compression algorithm that works well for streams of data where sequences of the same value (so called *runs*) occur. A simple example is the string "1111111222222333333". Using RLE, this string would be represented as "716263", indicating that the stream consists of seven ones followed by six two's and then six three's. When we generate our traces, the occupied C-state is used as the value, and the time spent in the power state is the sequence length. Updating the C-state trace of a CPU-core thus amounts to the constant time operation of appending these two numbers to a buffer. Whenever the buffer is full, it is copied and sent to the C-state selection/update thread using message passing. We employ CPU-state traces with microsecond granularity. When compared to storing a non-compressed stream of integers at microsecond fidelity, the use of RLE results in a a compression rate of over 99% for real traces corresponding to low, normal, and heavy load respectively. The cost of sending RLE C-state traces is shown in figure 4.9, and increases linearly with trace length.

Selecting the Best C-state

The C-state expert update thread is the component that calculates which C-state to use at a CPU-core at any given time. There is only one such thread, and it manages the weights of all experts for all available CPU-cores in the system. It receives the run-length encoded power state traces for the different cores asynchronously via message passing. When this happens, it will trigger the recalculation of the weights and best performing C-state for the core that

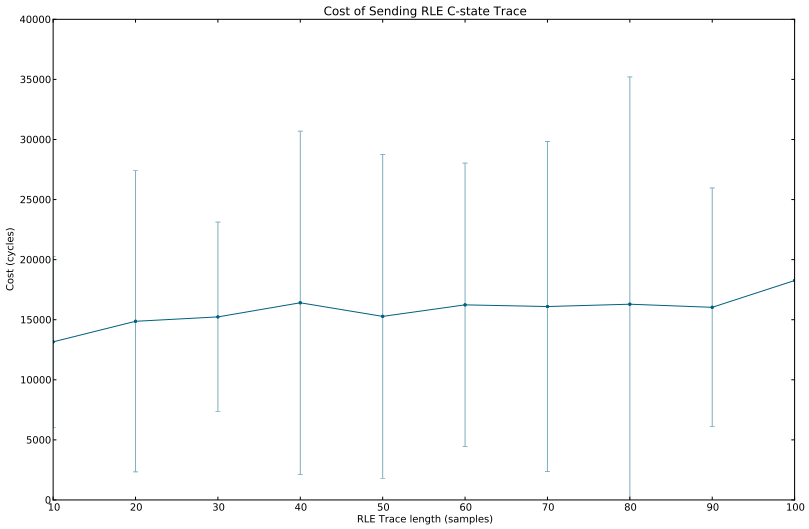


Figure 4.9: Average cost of sending a RLE C-state traces of varying lengths. The error bars show the standard deviation. All numbers are calculated from 10 000 measurements.

sent the trace.

The reason for handling weight updates in a separate thread is twofold. First, the complexity of the idle function is kept low, and it is only loosely coupled to the expert selection algorithm. Second, and more important: floating-point arithmetic can only be performed in the context of a thread. Vortex uses lazy saving of floating point registers, instead of saving them on each context switch[7]. This means that the SSE unit is set up to throw an exception on the first SSE instruction that is executed after each context switch. The exception can only be handled if a thread context is available, because the exception handler will save and restore floating point registers using this context.

The calculation of the best performing C-state and the updating of experts weights is performed according to the formal description of the Share Algorithm given in section 4.3. In our implementation, each of the experts correspond to using a single C-state only. Selecting the best performing expert thus implies selecting the best performing C-state. As described in section 4.3, the behavior of the Share Algorithm is tunable. Four different parameters provide flexibility with regards to how aggressively power is saved. For instance, the learning rate and share parameter makes it possible to adjust

7. Lazy saving of registers is an optimization that makes the common case, i.e. that floating point context is unnecessary, fast.

how sensitive the algorithm should be to changes in the workload, while the weighting parameters for wasted energy and added latency allows for adjusting which property is more important. This flexibility is desirable in several situations. For instance, in the face of a power failure, it could be possible to forsake latency service level objectives (SLOs) while maintaining availability by tuning the weights from a low latency setting and over to an aggressive power saving setting.

To facilitate the calculation of weights, support for mathematical functions such as *pow()* and *exp()* had to be added to the kernel. Typically, the exponential functions provided by standard math libraries are highly accurate. This accuracy, however, often comes at the cost of performance. In machine learning tasks, such as weight calculation in the Share Algorithm, approximations to the exponential function are often sufficient [80]. This allows much time to be saved. Because of this, we have implemented the approximation created by Schraudolph [80], and later modified by Cawley [17], and use it for weight calculation in our implementation of the Share Algorithm. The approximation works by manipulating the components within the IEEE-754 floating-point representation. This approach has the advantage of avoiding the need for a memory-consuming lookup table, as is commonly used in approximations to the exponential function. It is also significantly faster than lookup based implementations [80]. Our tests indicate a 12%–16% performance increase w.r.t. execution speed, when compared to the C standard math library.

The overhead of any PM algorithm is critical, as too expensive algorithms might result in reduced energy savings. We have measured the cost of executing a weight update with our implementation of the Share Algorithm, and concluded that the computational overhead (in the order of 5–20 microseconds) is negligible. The length of the RLE trace, as well as the number of experts are the key factors determining this cost. Figure 4.10 shows the measured overhead for varying RLE trace lengths and different numbers of experts. The figure shows that the necessary calculation time increases linearly with trace length. The cost of additional experts is moderate for small numbers. According to [39], the Share Algorithm performs nearly as good with 10 experts as with 100, so measurements with more than 10 experts are deferred. Further, no more than a single expert is necessary per C-state presented by the platform, so in practice the number of experts is low.

When leaving the idle function, a critical issue w.r.t. latency is that of sending RLE C-state traces to the update thread. How often this happens is a trade off between responsiveness, and the cost of sending the trace. If the traces are shorter, a new *best* C-state can be computed more often. The penalty, is that the frequency of relatively expensive send operations increases as well. The common case of not sending a trace is inexpensive: a constant time

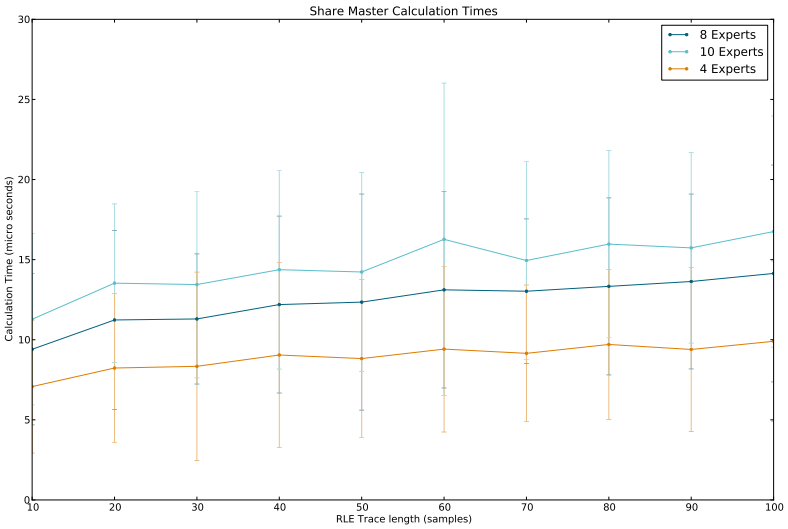


Figure 4.10: The computational overhead of recalculating the Share Algorithm expert weights. Notice that these numbers include the receiving of the RLE trace structures, as well as the cost of freeing all associated buffers. The graphs shows means and standard deviations of 100 000 measurements.

operation updating the RLE C-state trace is all that is necessary. Figure 4.11 shows the dynamic selection of best performing C-state for different cores in the system. Because the different cores execute different loads, they select their appropriate (best performing) C-state independently. Code listing 4.4 contains an excerpt of the idle function modified to support the entering of the best performing C-state, as well as the generation and sending of RLE C-state traces.

```
1  void idle(void)
2  {
3      /* Initialization */
4          .
5          .
6
7      // Enable interrupts
8      _INTERRUPT_ENABLE_UNCONDITIONAL();
9
10     // Loop till something happens
11     while (IDLEFLAG == TRUE) {
12
13         // Ensure power management components activated
14         if (cpulocal[CPU_ID].cl_pm_active) {
15
16             // Update RLE C-state trace
17             rle_add_sequence(cpulocal[CPU_ID].cl_core_state_rle, (int64)CSTATE_TYPE_Co, (int64)(arch_timestamp_usec()
                       - cpulocal[CPU_ID].cl_previous_state_change));
18             if ( rle_is_full (cpulocal[CPU_ID].cl_core_state_rle)) {
19
20                 rle_copy = copy_rle(cpulocal[CPU_ID].cl_core_state_rle);
21                 enqueue_rle(CPU_ID, rle_copy);
22
23                 // Reset local rle so we can continue using it
24                 reset_rle (cpulocal[CPU_ID].cl_core_state_rle);
25             }
26
27             // Sample timestamp so we can log how long the core was halted
28             cpulocal[CPU_ID].cl_previous_state_change = arch_timestamp_usec();
29
30             // Enter C-state according to what performs best
31             if (IDLEFLAG == TRUE) {
32                 mwait_idle_with_hints(NEXTENSIONS, cpulocal[CPU_ID].cl_cstate_hint);
33             }
34
35             // Update RLE C-state trace
36             rle_add_sequence(cpulocal[CPU_ID].cl_core_state_rle, (int64)cstate, (int64)(arch_timestamp_usec() - cpulocal[
                       CPU_ID].cl_previous_state_change));
37             cpulocal[CPU_ID].cl_previous_state_change = arch_timestamp_usec();
38             if ( rle_is_full (cpulocal[CPU_ID].cl_core_state_rle)) {
39
40                 rle_copy = copy_rle(cpulocal[CPU_ID].cl_core_state_rle);
41                 enqueue_rle(CPU_ID, rle_copy);
42
43                 // Reset local rle so we can continue using it
44                 reset_rle (cpulocal[CPU_ID].cl_core_state_rle);
45             }
46         }
47     }
48
49     // Disable interrupts
50     _INTERRUPT_DISABLE_UNCONDITIONAL();
51
52     if (DEBUG_CPUMUX_IDLE)
53         syslog(LOG_DEBUG, "Idle on core %d resuming cpumux", CPU_ID);
54
55     // Dispatch CPUMUX thread
56     arch_thread_dispatch(&cpumux_tcbref[CPU_ID]);
57
58     while (1) ;
59 }
```

Listing 4.4: Idle loop modified to support entry of best performing C-state.

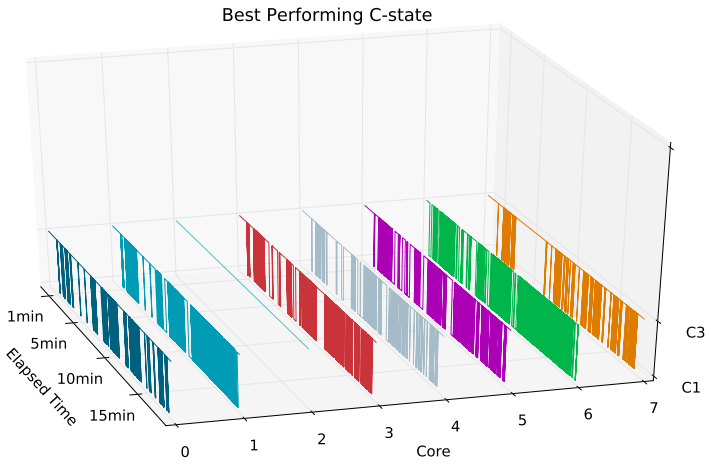


Figure 4.11: The figure shows how different C-states are found to be performing the best at runtime. Notice that during the generation of this trace, only C1 and C3 was used.

4.5 Per-core Performance State Management

Performance management of CPUs concerns the problem of selecting the correct CPU frequency for the current workload. In essence, this means reducing the core operating frequency when the load is low, and increasing it under periods of higher load. Because of the quadratic relationship between power consumption, voltage, and frequency described in section 2.4.3, previously proposed solutions for reducing power consumption have involved running jobs as slowly as possible while still respecting quality of service (QoS) [82], [72]. While this kind of approach has proven itself useful—and certainly still can be—modern CPUs differ significantly in architecture and design from those only a few years old. On more modern CPUs, the power consumed when in the idle state is significantly lower than on older models. If the power consumption in the idle state is low enough, net power savings can be achieved by running the CPU at full speed when there is work to be done. By finishing the work quickly, it is possible to maximize the time idle. This phenomenon is called “race to halt”, or “race to idle”. In the Linux 3.9 kernel, Intel CPU P-state drivers were introduced to take advantage of exactly this [7]. Throughout the rest of this section we describe the implementation of our per-core performance state management algorithms.

4.5.1 Quantifying CPU Utilization

Quantifying the utilization of a CPU-core can be difficult. As pointed out in previous work, task execution speed is often to some extent limited by I/O accesses or the speed of the memory bus [51], [50]. While keeping the CPU frequency high ("racing to idle") is efficient when the workload is CPU-intensive, this is not true if the workload is, for instance, bound by the speed of the memory bus. This situation is sometimes referred to as *performance saturation*, which simply means that the system is unable to make use of all the available clock cycles [51].

In our implementations, we follow the approach taken by Isci et al. [45] to measure the "memory boundedness" of the currently executing workload. More specifically we use PMCs to obtain the number of last-level cache (LLC)-misses and the number of μ-ops[8] retired per CPU-core. The ratio between these, hereafter referred to as "mem/μ-ops" is used to classify the workload into different *phases*. The choice of this metric is based on it being invariant to the frequency at which the core is running [45]. This property is strictly necessary, as our performance adjustment algorithm will employ dynamic voltage scaling (DVS) to increase and decrease the performance level of the cores, thus altering their frequency in the process.

We use "mem/μ-ops" to classify the workload into different phases similarly to [92] and [45]. While the first of these works calculates DVS settings for different application regions based on a formulation of performance loss, the latter translates these measures to "mem/μ-ops" rates. We borrow from [45] for our phase classifications, but make necessary changes to accommodate our hardware. Specifically, we determine the thresholds for phase classifications experimentally by running various workloads, both memory and CPU-bound. The "mem/μ-ops" to phase mappings are listed in table 4.1. Phase 1 corresponds to a highly CPU-bound phase of execution where it is desirable to "race to idle". Phase 2 matches mixed loads, while phase 3 corresponds to a highly memory bound phase where performance saturation is occurring, and the CPU frequency can be reduced without significant loss in performance.

4.5.2 Global Phase History Table Predictor

We implement our first per-core performance management functionality based on the use of a Global Phase History Table (GPHT) predictor. This approach is similar to that of [45], and based on a branch prediction technique which

8. In computer CPUs, micro-operations (μops) are detailed low-level instructions used to implement complex machine instructions [33].

"mem/μ-ops"	Phase Number	P-state
< 1.25	1 (race to idle)	0
$[1.25, 1.75]$	2	1
> 1.75	3 (highly memory-bound)	2

Table 4.1: Mappings from "mem/μ-ops" ratios to execution phases.

has proven to perform very well [94]. The technique, called *Two-level Adaptive Branch Prediction* uses two levels of history to make decisions, the n last branches that have been encountered, and the recent behavior for the specific patterns of these n branches. This is a online statistical machine learning technique, and it relies on the existence of recurrent patterns in execution. Although designed for branch prediction in CPUs, the concepts used can be directly translated to the power management domain; instead of capturing history regarding the recent branches and which of these to take, information about recent performance states and what frequency to use next can be stored. One of the key properties of the GPHT predictor is that it is completely agnostic to the workloads being executed. No analysis of the workloads or training of the algorithm has to be performed. It thus lends itself naturally to environments where workloads can exhibit high variability and change over time.

The Global Phase History Table is illustrated in figure 4.12, and consists of the following components:

- **Global Phase History Register (GPHR):** A global shift register that tracks the last n observed phases (as determined by their "mem/μ-ops" ratios). The length of the history is given by the *GPHR depth*. At each sampling point, the GPHR is updated, and it functions as a sliding window containing the n last samples. At each sample point, the contents of the GPHR is extracted and used as a key to index a Pattern History Table.

- **Pattern History Table (PHT):** The PHT contains a number of previously observed patterns (keys from the GPHR) and their predicted next phase. These patterns and predictions are updated using a least recently used (LRU) scheme to avoid having the table grow indefinitely. If a key from the GPHR is missing (it has not been encountered before or has been evicted from the table), the last observed phase is stored as the prediction. At the end of each sampling period, the actual phase that has been observed is inserted in the PHT. In this manner, the Global Phase History Table predictor will do a lookup on the recent history, and

make a prediction corresponding to the phase that was last observed succeeding that exact pattern.

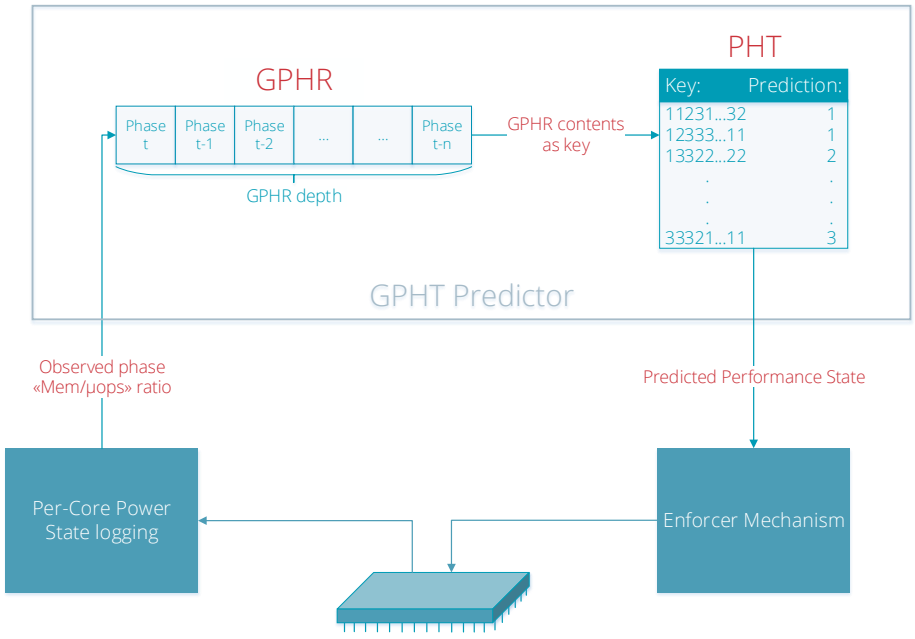


Figure 4.12: The implementation of the Core Performance State Manger.

All components of the GPHT predictor are implemented within the Vortex kernel. The GPHR is implemented using a circular buffer, while the implementation of the PHT is more involved. To achieve $O(1)$ complexity for retrieving and updating predictions, a dictionary is used to allow efficient mappings from GPHR obtained keys to predictions. To allow for efficient LRU eviction from the PHT, the actual predictions are stored in a LRU-queue implemented using linked data structures. This is illustrated in figure 4.14. As the retrieval and updating of predictions are operations that are performed frequently, it is critical that they are inexpensive. Figure 4.13 displays the cost of performing these actions. The plots are created from 10 000 data samples. However, note that for the sake of visualization, only values within the 99th percentile are included. As can be seen from these plots, the costs of the operations are low; a lookup and update operation costing on average only ~2600 and ~4000 cycles respectively.

The sampling, phase prediction retrieval, updating of predictions, and performance state transitions are all controlled from a thread (see code listing 4.6). There are several reasons for this choice:

- Since performance management is performed on a per CPU-core basis,

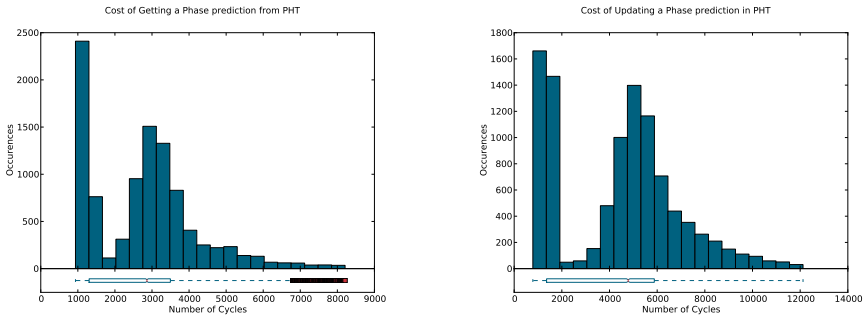


Figure 4.13: The above plots show the cost distribution for the retrieval and updating of predictions from the PHT. 10 000 measurements were used as basis for each of the plots, but for the sake of visualization only values within the 99th percentile are included.

using threads is a simple way to avoid having to protect the GPHT data structures with locks. Because each core runs a separate thread, there can exist no race conditions.

- The classification of phases relies on high precision floating point operations. Because of the need for accuracy, fixed point approaches cannot be used[9]. As explained in section 4.4.3, only threads have the necessary context to perform floating point arithmetic.

- Sampling of the PMCs that define different execution phases must be done periodically. If, for instance, the PMCs were sampled when entering or exiting the idle state (a scheme similar to that used for per-core power management), it would become impossible to increase the CPU frequency when utilization is growing. A steady increase in the amount of work would result in the core always being busy, never entering the idle loop. This, in turn, would keep the core operating at the frequency it was running at before the influx of work occurred. Threads lend themselves naturally to running periodic tasks, as they can simply be suspended in between sampling points.

However, the choice of using threads is not without problems. The fact that the thread will perform work periodically can result in spoiled opportunities for energy savings if not handled properly; there are situations where the performance management threads might wake up halted or sleeping CPU cores.

9. Fixed point arithmetic libraries can be used to emulate floating point arithmetic. Such libraries are often used either because the processor lacks support for floating point operations, or to increase performance [89]. However, this performance comes at the cost of accuracy.

```
1
2  void idle(void)
3  {
4      /* Initialization */
5          .
6          .
7
8
9      // Ensure power management components activated
10     // If not, busy wait
11     if (pm_is_active) {
12
13         //Start by suspending execution of performance state selection thread
14         if (thread_active) {
15
16             if (VxO_REFLOCK(&pstate_timer_lock, &cpulocal[CPU_ID].cl_pstate_selector_thread_ref)) {
17
18                 tcb = (tcb_t*) VxO_REFGETOBJ_NOCHECK(&cpulocal[CPU_ID].cl_pstate_selector_thread_ref);
19
20                 timer = (timer_t*) VxO_REFGETOBJ_NOCHECK(&tcb->tcb_resumetimer);
21                 scheduled_timeout = timer->ti_timeout;
22                 vxerr = thread_timer_clear(&cpulocal[CPU_ID].cl_pstate_selector_thread_ref);
23
24                /* Error handling code */
25                     .
26                     .
27
28             }
29             VxO_XUNLOCK(&pstate_timer_lock);
30         }
31
32         // Enter sleep state with interrupts enabled
33         _INTERRUPT_ENABLE_UNCONDITIONAL();
34         mwait_idle_with_hints(NEXTENSIONS, cpulocal[CPU_ID].cl_cstate_hint);
35         _INTERRUPT_DISABLE_UNCONDITIONAL();
36
37     } else {
38
39         //Busy wait with interrupts enabled
40         _INTERRUPT_ENABLE_UNCONDITIONAL();
41         while (IDLEFLAG == TRUE) {
42         }
43         _INTERRUPT_DISABLE_UNCONDITIONAL();
44     }
45
46
47     //Resume execution of performance state selection thread
48     if (pm_is_active && thread_active) {
49         if (VxO_REFLOCK(&pstate_timer_lock, &cpulocal[CPU_ID].cl_pstate_selector_thread_ref)) {
50
51             tcb = (tcb_t*) VxO_REFGETOBJ_NOCHECK(&cpulocal[CPU_ID].cl_pstate_selector_thread_ref);
52
53             timeout_value = MAX(scheduled_timeout - arch_timestamp_usec(), 1);
54             vxerr = timer_post_fmt(&tcb->tcb_resumetimer, timeout_value, thread_resume, "rI", (objref_t *)&cpulocal[
                       CPU_ID].cl_pstate_selector_thread_ref, THREAD_FLAG_TIMEOUT | THREAD_FLAG_SUSPENDBALANCE);
55
56             /* Error handling code */
57                     .
58                     .
59
60         }
61         VxO_XUNLOCK(&pstate_timer_lock);
62     }
63
64     // Dispatch CPUMUX thread
65     arch_thread_dispatch(&cpumux_tcbref[CPU_ID]);
66
67     while (1) ;
68 }
```

Listing 4.5: Idle function modified to support P-state selection.

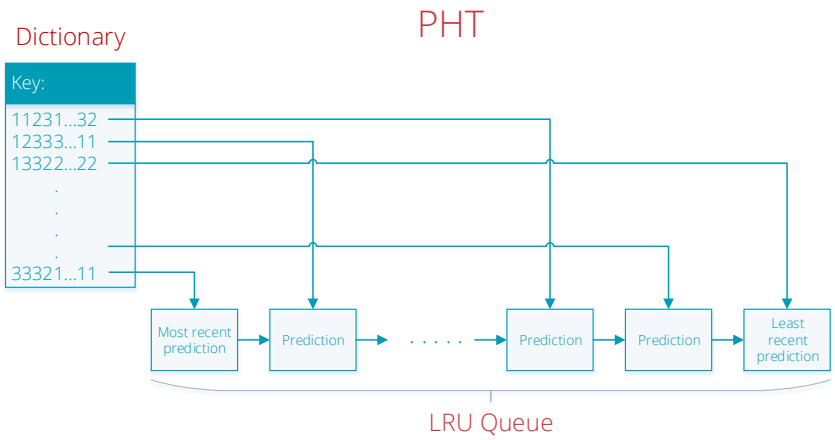


Figure 4.14: The figure illustrates the implementation of the GPHT predictor. Due to the use of both a dictionary and LRU-queue, $O(1)$ complexity is achieved for both prediction lookups and updates.

To avoid such unnecessary wakeups, we add code on entry and exit from the idle function that simply suspends and reschedules the thread (see code listing 4.5). This of course adds latency, but our tests show the overhead to be negligible (see figure 4.15). In addition, there is a cost associated with running the thread periodically. Again, this cost is in the order of ~5000 to ~30 000 cycles, with an average cost of ~15 000. Out of these, the cost of updating and querying the GPHT predictions amount to 26% and 17% of the cycles on average. The remaining 57%, or approximately 9000 cycles, include the reading of PMCs, phase classification, and loop overhead.

The enforcement module is the final component of the per-core performance management scheme. The transitions between different CPU P-states is performed using ACPI-objects presented by the platform. A detailed description of this process is given in appendix A.

Figure 4.16 illustrates how different P-states are chosen by cores according to their experienced load. For a closer description of this workload, see section 5.1.4. Under this workload ApacheBench (AB) is run with a workload trace corresponding to high load (see section 5.1.3 for details). This results in Apache worker threads being dispatched to all cores. This is a memory bound task, as indicated by the cores seldom entering P0. Core 5, in addition to serving Apache worker threads, runs a compute intensive task that calculates prime numbers and sleep repeatedly. This is clearly visible by the core entering P0 whenever the compute intensive task is scheduled. Core 3 runs the same cpu intensive task as core 5, but for shorter durations. This

```
1  vxerr_t pstate_selector_thread(void)
2  {
3      /* Initialization */
4              .
5              .
6
7      while (1)
8      {
9
10         //Make prediction and clock CPU core according to this
11         predicted_phase = gpht_predictor_get_prediction(cpulocal[CPU_ID].cl_gpht_predictor);
12         if (predicted_phase != gpht_predictor_get_current_phase())
13         {
14             vxerr = gpht_predictor_change_performance_state(predicted_phase);
15
16             /* Error handling code */
17                     .
18                     .
19         }
20
21         //Put the thread to sleep for desired interval
22         thread_suspend_timeout(&cpulocal[CPU_ID].cl_pstate_selector_thread_ref, PSTATE_SELECTOR_SLEEPTIME);
23
24         //Read out performance counter registers and update predictor accordingly
25         mem_to_uops_ratio = llc_miss_to_uops_ratio();
26
27         observed_phase = predictor_classify_ratio(mem_to_uops_ratio);
28
29         vxerr = gpht_predictor_update_predictor(cpulocal[CPU_ID].cl_gpht_predictor, observed_phase);
30
31         /* Error handling code */
32                 .
33                 .
34
35     }
36 }
```

Listing 4.6: Performance State Management Thread

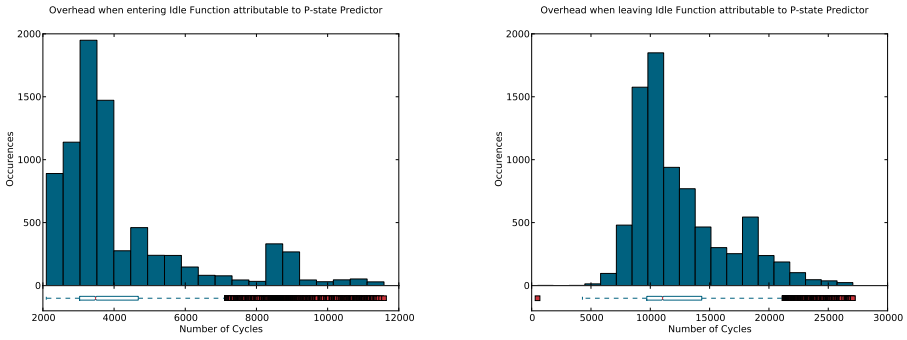


Figure 4.15: The above plots show the overhead added to the entry and exit of the idle function following the introduction of per-core performance state management. The plots show averages over all cores, where 1000 measurements have been obtained for each. For the sake of visualization only values within the 99th percentile are included.

Description	Avg. # Cycles	Time (μs)		
		2.0GHz	2.33GHz	2.66GHz
Get prediction from PHT	2628	1.31	1.13	0.99
Update prediction in PHT	4000	2.00	1.71	1.50
One iteration in GPHT	15456	7.73	6.63	5.82
Idle function entry overhead	4214	2.11	1.81	1.58
Idle function exit overhead	12434	6.22	5.34	4.67

Table 4.2: Summary of Core Performance Management costs and overheads. The table shows averages over all cores, where 10000 measurements have been obtained for each. For the sake of visualization only values within the 99th percentile are included.

results in the core only seldom reaching a combined workload that is CPU bound. Finally, Core 6 is busy serving interrupts originating from the network traffic.

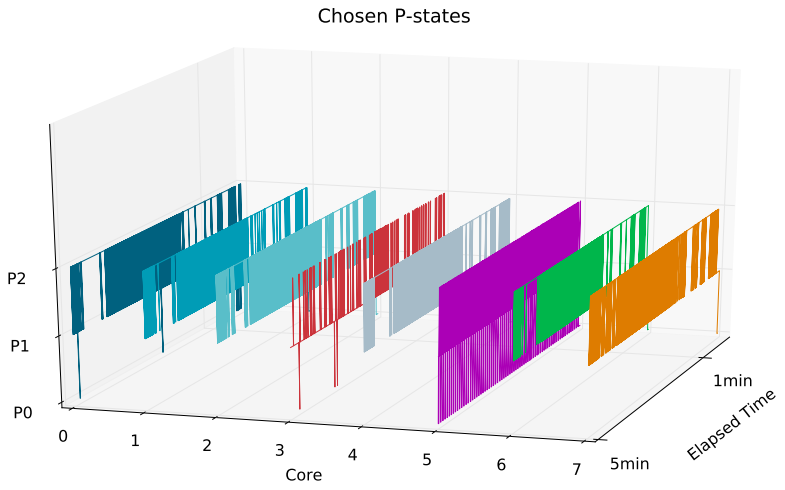


Figure 4.16: The figure illustrates how cores experiencing different loads select their P-states at runtime.

4.5.3 Naive Forecaster

In addition to the GPHT predictor, we also implement what is known as a *naive forecaster*. A naive forecaster simply predicts that the next phase will be

identical to the previous one.

We implement our naive forecaster with a single thread per core. This thread stores the observed phase, and uses it as the prediction for the next interval. When the interval is over, it observes the actual mem/μ-ops ratio from the interval, classifies it as a phase, and stores it. A code excerpt showing this thread is given in listing 4.7. Similarly to the GPHT predictor, the idle loop is modified to stop the thread whenever a C-state is entered, and resume it when execution continues.

Figure 4.17 shows a plot of the cost of running the naive forecaster loop one iteration. The cost is substantially lower than for the GPHT predictor.

```
1  static vxerr_t naive_forecast_pstate_selector_thread(void)
2  {
3      /* Initialization */
4              .
5              .
6
7      while (1) {
8
9          predicted_phase = observed_phase;
10         if (predicted_phase != gpht_predictor_get_current_phase()) {
11             vxerr = gpht_predictor_change_performance_state(predicted_phase);
12         }
13
14         //Put the thread to sleep for desired interval
15         thread_suspend_timeout(&cpulocal[CPU_ID].cl_pstate_selector_thread_ref, PSTATE_SELECTOR_SLEEPTIME);
16
17         //Read out performance counter registers and update predictor accordingly
18         mem_to_uops_ratio = llc_miss_to_uops_ratio();
19         observed_phase = predictor_classify_ratio(mem_to_uops_ratio);
20     }
21 }
```

Listing 4.7: Naive Forecaster.

4.6 Energy Efficient Scheduling

Energy efficient scheduling is the last of the three axes we defined for CPU PM. Scheduling algorithms have been used to reduce power consumption in various domains. For instance, to schedule VMs at different physical hosts in clouds [55]. Much work has also gone into energy efficient scheduling in real time systems [38], [72], [95], where DVS is employed while still meeting deadlines. Other approaches exploit knowledge about the platform topology, by attempting to keep as many components as possible powered down at all times [83]. In this section, we describe the implementation of the energy efficient scheduling algorithm employed in ROPE.

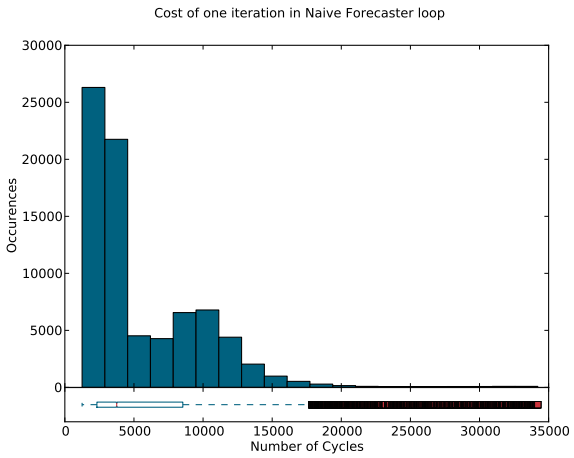


Figure 4.17: Overhead of naive forecaster.

4.6.1 Topology Agnostic Dynamic Round-Robin

This section describes the implementation of a topology agnostic algorithm for placing work on CPU-cores. The intuition behind the algorithm is that if as few cores as possible are active at any given time, power can be saved. The algorithm is naive in that it does not consider the topology of the platform. This means that no effort is made to keep caches warm, or to limit the number of packages with active cores.

The algorithm is hooked directly into the existing scheduling framework in Vortex. More specifically, it is executed as a *load balancing* action, which is invoked periodically. The algorithm attempts to limit the number of cores in use by filling one and one core, only adding new cores to the current working set whenever the average load reaches a given threshold. In dynamic round-robin (DRR), CPU-cores can be in one of three states:

- **Active:** Tasks are scheduled on active cores in a round-robin fashion.
- **Retiring:** Cores that are retiring can not receive new tasks, but will continue running tasks that have been assigned to the core already. Over time, tasks will be migrated away from cores that are in the retiring state. After being in the retiring state some predefined amount of time, a core will enter the inactive state.
- **Inactive:** Tasks are not scheduled on inactive cores.

In [55], the authors use a similar classification of VMs to perform scheduling

of VM instances. For convenience, we chose to use the same terminology, although our solution differ from theirs in how we enforce and chose our thresholds.

Figure 4.18 shows an illustration of the structure of the dynamic round-robin (DRR) scheduler. As mentioned, tasks are scheduled on the active cores round-robin. If no cores are active, or the average utilization of all active cores is greater than some threshold Θ_{recruit}, additional cores are added to the set of active cores. When additional cores must be recruited, cores in the retiring state are targeted first. This is because they are less likely to have entered a sleep state, and waking a sleeping core has a greater associated latency than using one that is already active. Only in the case that no cores are retiring, will inactive cores be awoken and added to the set of active cores.

Cores enter the retiring state whenever their utilization drops below some threshold, Θ_{retiring}. Cores remain in this state a predefined amount of time, T_{retiring}, before entering the inactive state.

Using the three parameters Θ_{recruit}, Θ_{retiring}, and T_{retiring}, the behavior of the scheduler can be tuned. The value of Θ_{recruit} controls the number of cores that will be activated for a given amount of load, and is defined as:

$$\Theta_{\text{recruit}} = \frac{1}{m} \sum_{i=1}^{m} C_i \ , C \in \text{ Active Queue} \tag{4.7}$$

For instance, a value of $\Theta_{\text{recruit}} = 90$ means that the average load over all active cores must reach 90% before another core is moved into the active set. The Θ_{retiring} parameter determines how sensitive the active set is to fluctuations in the workload. A high value will result in cores moving between the active and retiring state frequently, while a low value will keep the set of active cores more stable. Finally, T_{retiring} will determine how long cores remain in the retiring state. If the value is too large, cores are less likely to enter the inactive state and power may be wasted. However, too small values might also be wasteful as the result will be frequent state transitions.

Figure 4.19 shows the utilization of different cores over time as the total system-wide load increases. The figure shows how cores are recruited gradually, and only when cores reach a level of utilization that warrants this (Θ_{recruit}). The very low levels that are observable for some of the cores originate from interrupt- and message processing[10]. Our scheduler only consider threads, and does not control these activities.

10. Recall that all communication in Vortex is via asynchronous message passing.

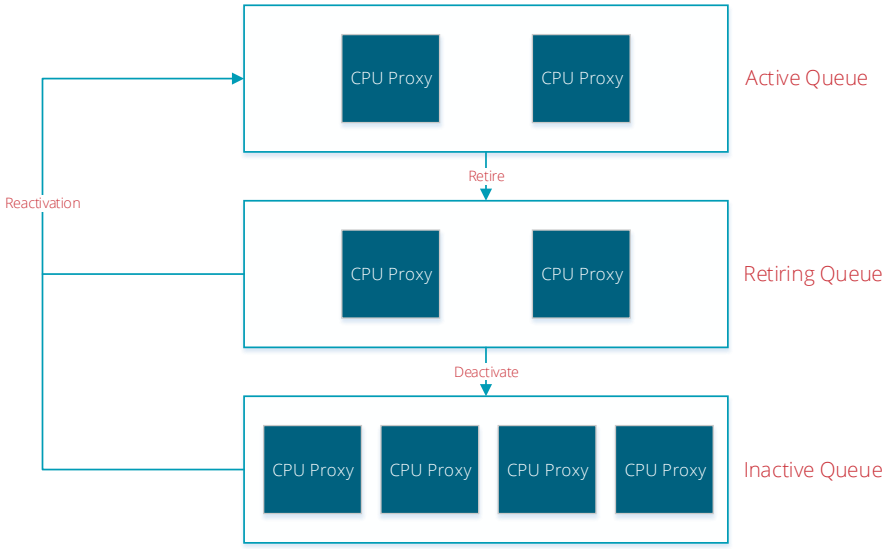


Figure 4.18: Implementation of energy efficient scheduler.

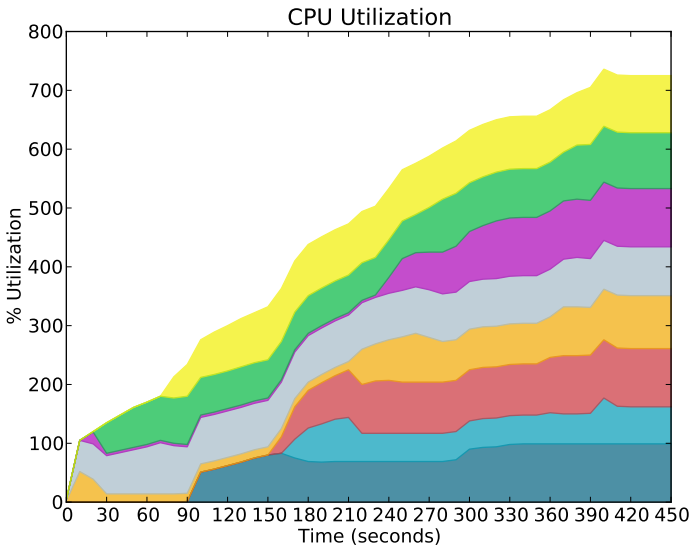


Figure 4.19: CPU utilization over time under dynamic round-robin scheduling.

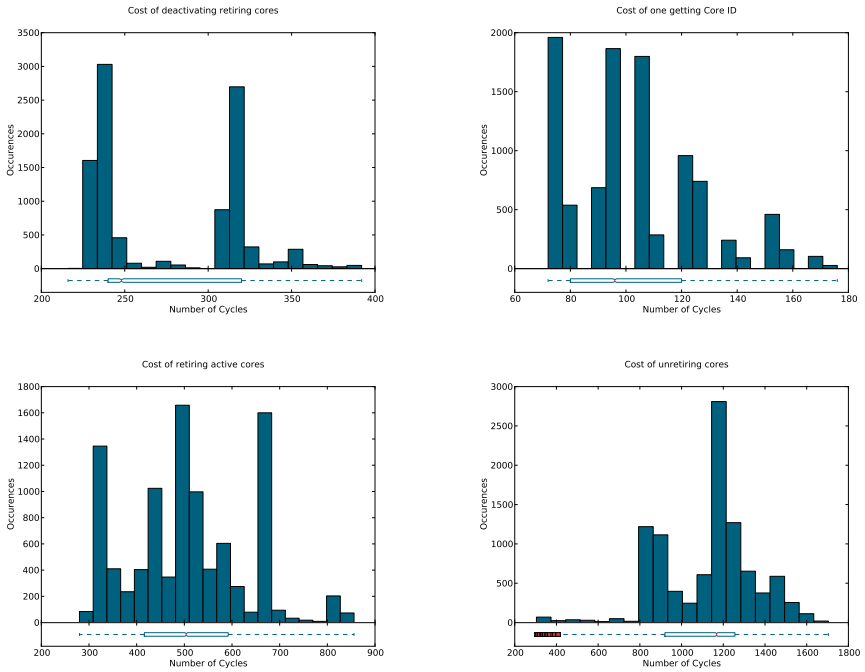


Figure 4.20: The above plots displays overheads for different operation within the energy efficient scheduler implementation. Note that for the sake of visualization, only values within the 99th percentile are included. Note that all the graphs are based on 10 000 measurements.

Figure 4.20 contains plots detailing the cost of various operations in the topology agnostic round-robin scheduler. All operations are cheap, and the most expensive, unretiring cores, costs less than 1200 cycles on average.

5 Evaluation

In this chapter we evaluate ROPE and its PM policies. We start by describing our experimental methodology, platform, and workloads. Following this, we will evaluate C-, and P-state management policies, before turning our attention to energy efficient scheduling.

5.1 Methodology

Throughout the evaluation of our PM-policies, we will use both internal and external measures. While internal measures, such as predictor accuracy, are certainly interesting, external measures such as power consumption and user-perceived performance are the ones that really matter. Based on this, we evaluate our policies and algorithms using workloads designed to mimic real-life usage, but also run experiments that highlight the behavior of our policies through internal measures.

When evaluating the efficiency of PM-policies in the setting of cloud computing, the amount of power saved is not the only important factor: different SLOs govern how much performance that is necessary according to different metrics such as latency, throughput, and availability. A typical cloud service whose QoS is reliant on exactly these metrics, is web hosting. As a consequence, this is one of our services of choice when benchmarking our PM algorithms. In addition, we measure the performance characteristics of our

solutions using a industry standard database benchmark, TPC-C.

5.1.1 Experimental Platform

We run our experiments on a a set of Hewlet Packard ProLiant BL460c G1 blade servers. These blades are equipped with twin Intel Xeon 5355 processors running at a peak frequency of 2.66GHz, have 16GB of PC2-5300 DDR2 RAM, and a single 10K SAS hard drive. Note that seemingly identical blades might contain different hardware[1]. Thus, care has been taken to ensure identical hardware for all test systems. In addition, we use a set of similarly specced Dell PowerEdge M600 blade servers as compute nodes for generating load for our test systems. Each of the HP test systems are connected via a 1Gbit Ethernet link to a HP 1:10G Ethernet blade switch, while the Dell blades are connected to an Ethernet Pass-Through switch from the same vendor. In turn, these two switches are interconnected with 4×1Gbit connections. To avoid network bandwidth being an inhibiting factor, we limit the number of simultaneously benchmarked systems to 4, each of these being granted exclusive access to one of the 1Gbit links via the use of Virtual LANs (VLANs).

If not otherwise stated, all experiments including power consumption measurements are performed using a set of workload traces detailed in the following sections. For all web-based benchmarks, we run the server software within a lightweight VM OS providing the Linux 3.2.0 ABI [68], [69]. This, in turn, is run on top of Vortex. When executing workloads consisting of HTTP requests, we run an instance of Apache Web Server version 2.4. Database transaction workloads are run against a MySQL Community Server 5.6.17 instance.

5.1.2 Measuring Power Consumption

Even with homogeneous hardware, inter node variability is ever present. We measure the power consumption every second on a per node basis using the Intelligent Platform Management Interface (IPMI) protocol. The different blade servers used as test systems exhibit power consumptions of 182W–196W (a difference of over 5%), even when completely idle. When RAM from different vendors was used, the differences were even larger. Our tests indicate that the variance in power consumption is closely related to the physical device bays of the server enclosure. Thermal issues are unlikely to be the culprit as fan speeds throughout the entire enclosure vary only slightly.

1. For instance, our blades contains RAM from two different vendors. In addition, one of these were “green”, consuming considerably less energy than the other.

To compensate for this variability, all power consumption measurements are averaged over a minimum of 15 runs. These runs are in turn spread over at least 3 different nodes. By using this approach we avoid the scenario where different algorithms and configurations are given an unfair advantage or impediment simply as the result of node assignment. Averages also serve as a good measure for performance and energy consumption when working with large numbers of servers, as can be expected in a DC setting.

5.1.3 ApacheBench Workload Traces

To ensure that fair comparisons of different algorithms are possible, we evaluate these by executing a set of workload traces. These workloads are generated using ApacheBench (AB)[2], which is a program for testing web servers. AB allows us to measure performance in terms of throughput, availability, response times at different percentiles, and several additional metrics. To measure web server performance, AB can be run with different arguments. The arguments employed when generating our traces are as follows:

- URL: The URL to a file to be downloaded by AB.
- Number of requests: The number of requests AB should perform towards the web server.
- Concurrency: The concurrency level of the requests, that is, the number of simultaneous requests towards the web server.

In addition, we produce a spiky behavior by sleeping a randomly selected amount of time from within a predefined range. To assure that our PM algorithms are tested for varying workloads, we create three different traces corresponding to low-, normal-, and high load. For all of these, the files to be requested are selected randomly from a set containing files with sizes 1KB, 2KB, 4KB, 16KB, 32KB, 2MB, and 4MB. The number of requests, concurrency level, and sleep time are all selected randomly from within predefined ranges, as listed in table 5.1.

After these traces have been generated, they are stored so that they can be replayed. This is performed using a simple Python[3] script which simply reads each entry, sleeps the required amount of time, and executes AB with the parameters corresponding to the current entry.

2. http://httpd.apache.org/docs/2.2/programs/ab.html
3. https://www.python.org/

Workload	Description	Concurrency	Requests	Sleep time
Low	Very low and spiky load.	$[1, R]$	$[1, 50]$	$[1, 30]$
Normal	CPU load between 20%–30% and many idle periods shorter than one second. This corresponds to utilizations of real data centers [59], [15], [13], [12].	$[50, R]$	$[100, 500]$	$[0.1, 1.0]$
High	CPU load between 25%–30%. Longer periods of load, shorter idle periods.	$[100, R]$	$[500, 1000]$	$[0.01, 0.1]$

Table 5.1: Summary of AB Workload trace generation. All sleep times are listed in seconds. The R symbol is used to illustrate that the maximum possible concurrency is limited by the randomly chosen number of requests.

5.1.4 Prediction Test Workload

For some of our experiments, the AB traces described above are not suitable. In these cases, we use a somewhat different workload which is more variable. The reason we do this is that the AB traces are highly memory bound, and thus almost exclusively contain phases corresponding to the lowest performance state. This makes for a bad test when attempting to compare certain algorithms, for example ones that predict P-states. Instead we use a population of compute intensive threads (calculating prime numbers) with different characteristics spread over some of the cores, with a background load corresponding to the most work-intensive AB trace.

5.1.5 HammerDB and TPC-C Benchmark

We use the TPC-C database benchmark to measure intrusiveness and performance degradation resulting from our PM algorithms. TPC-C is an online transaction processing (OLTP) benchmark featuring multiple transaction types, and complex database- and execution structures.

TPC-C is approved by the Transaction Processing Performance Council (TPC), and simulates a complete environment where a population of terminal operators executes transactions against a database. The benchmark is centered

around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at multiple warehouses [3].

We use HammerDB[4] to execute TPC-C against our test servers. HammerDB is a tool that provides a simple graphical interface for configuration and benchmarking of databases[5]. Figure 5.1 shows a screenshot from a benchmarking session. HammerDB provides two measures of performance, the number of transactions per minute (TPM), and the number of new-order transactions per minute (NOPM), which is the performance metric of TPC-C. For all tests using TPC-C and HammerDB, we use a configuration of 5 warehouses and 5 virtual users. We sample data from at least 3 separate physical hosts, and calculate averages and standard deviations from a minimum of 15 measurements.

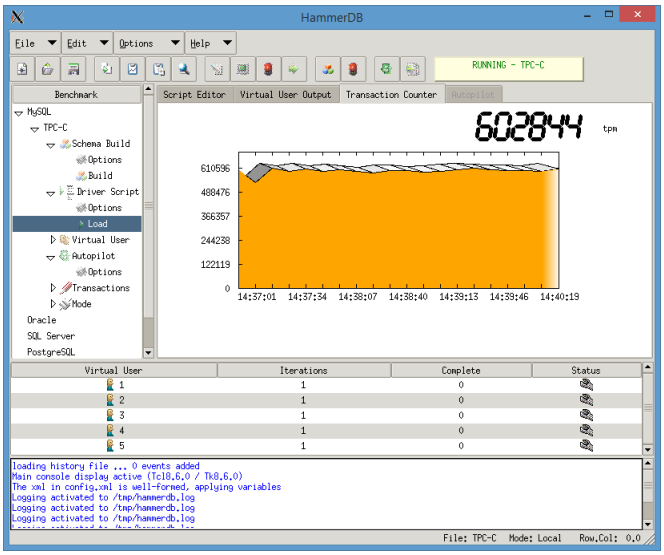


Figure 5.1: Screenshot of running TPC-C with HammerDB.

5.1.6 Fine Grained Workload

Our final workload allows fine-grained control over the total load in the system. This is especially useful when evaluating our energy efficient scheduling algorithm.

4. http://hammerora.sourceforge.net/
5. HammerDB currently supports Oracle, Microsoft SQL Server, MySQL, PostgreSQL, and Redis databases.

The workload is generated by spawning new CPU-intensive threads, each corresponding to a configurable amount of CPU consumption. This is done until a global (also configurable) limit for CPU load is reached. For example, new threads, each using 5% of the compute resources of a core, can be spawned until the total system-wide CPU load is 500% (corresponding to five cores being fully utilized).

The threads generating the actual load are busy waiting and sleeping repeatedly in a pattern that sums to the total configured CPU consumption.

5.2 CPU Power State Management

In this section, we describe our experiments involving policies for managing CPU C-states. When measuring power consumption and total completion time, we sum the samples over the entire trace. This is done to better illustrate the differences between different policies. We measure instantaneous latency, as perceived by a user, per request. For each workload we also measure the power consumed and experienced latency when not using any PM at all. These measurements serve as baselines for comparing the different policies.

5.2.1 Aggressively Entering C-states

Aggressive entry of different C-states is the simplest of our policies for managing CPU C-states. Our platform supports a total of six different C-states (as enumerated by the use of the CPUID instruction). These are C1, C2, and C3—each with two sub-states. We were unable to measure any difference between sub-states belonging to the same major C-state. Further, neither CPUID nor any freely available documentation on our CPUs provide any information about the latencies of entering these states[6]. For simplicity, we use only the deepest C-states. Notice that for C1, this corresponds to the C1E state, which is entered automatically by our CPUs if all cores on a chip are halted.

Figure 5.2 shows the measured power consumption of entering various C-states as soon as an idle period occurs. As is clear from the figure, enabling direct entry of any C-state results in significant energy savings. Because of the large standard deviations, it is impossible to separate entering C1 or C2 from each other with any degree of certainty. However, entering C3 directly

6. ACPI does provide some latencies, but these do not correspond with the ones used by Ubuntu Linux 12.04 when installed on our platform. We use the largest values, assuming that these correspond to the deepest sub-states.

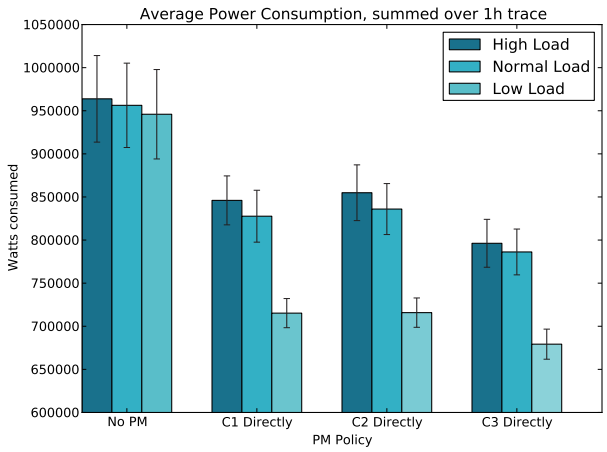


Figure 5.2: Mean power consumed over 1 hour long AB web traffic traces. The amount of power consumed is summed over the entire trace to better show differences between the different C-states. The error bars show standard deviations.

seems to consume less energy on average.

Although a substantial amount of energy is saved for all workloads, the relative difference compared to using no PM is especially large for the low load. This is because this workload trace features relatively long idle periods. As described in section 2.2, such workloads allow for considerable reductions in energy consumption as the cost of transitioning in and out of different power states is by far outweighed by the energy savings. The relative savings for different workloads are displayed in table 5.2. As can be seen, the relative energy savings is over 4% better for entering C3 directly, than if C2 or C1 is entered.

Workload Relative Energy Savings

Workload	C1	C2	C3
Low	24.4%	23.3%	28.2%
Normal	13.5%	12.6%	17.8%
High	12.2%	11.3%	17.4%

Table 5.2: Relative energy savings for different workloads when entering C-states directly. The percentages are calculated from the means shown in figure 5.2.

The loss of performance resulting from the use PM policies is also important when deciding on a policy. From logs obtained from runs of our AB workloads, we calculate the mean excess completion time for different policies. For each

of the 1 hour traces, we sum the time before 99% of requests are served. Again, the summation is performed to make any difference easier to detect. Figure 5.3 shows the mean excess completion time when entering C-states directly, i.e., the difference between the use of a policy and no PM. Interestingly, entering C1 seems to increase the completion time slightly on average, while it is difficult to say anything in particular for entering C2 or C3. Our conclusion is that entering C-states directly results in close to no increase—but does induce some variability—in total completion time. Note that this summed completion time does not correspond to added *instantaneous* latency, as experienced by a user issuing a request. Table 5.3 shows the relative increase in completion time and variability for entering C-states directly, as opposed to using no PM.

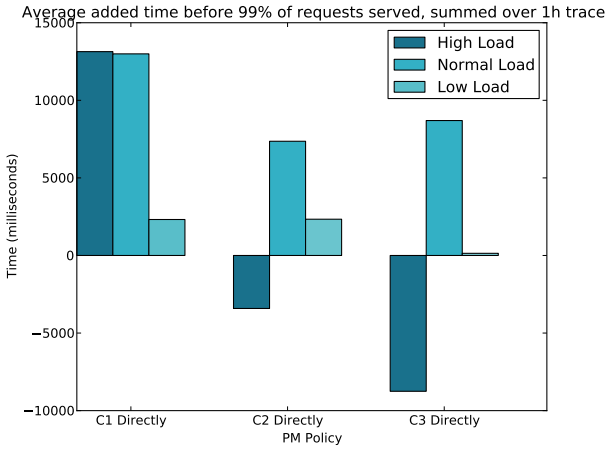


Figure 5.3: The figure shows the added completion time induced by aggressively entering different C-states. The measure of added completion time is time before 99% of requests are served, and the plots show the difference between entering C-states aggressively and no PM.

To evaluate instantaneous latency, we examine the AB logs on a per-AB-run basis. For each workload and policy, we calculate the mean for each of the individual AB runs (over all the traces), and calculate the difference from the mean obtained when using no PM policy. Because the different AB traces contain a varying number of requests, we normalize the difference per request. These differences are shown in figure 5.4. Notice that even though the length of the workloads in terms of AB runs are different, they all have the same 1 hour duration. Negative values would correspond to execution with PM being faster than running the system at maximum performance. Clearly, these measures are artifacts. For each AB run, all outliers, $|x| > 2\sigma$, σ being the standard deviation, are removed to reduce the amount of noise. We see that the added latency only very rarely exceed one millisecond, and most of the time is close to zero.

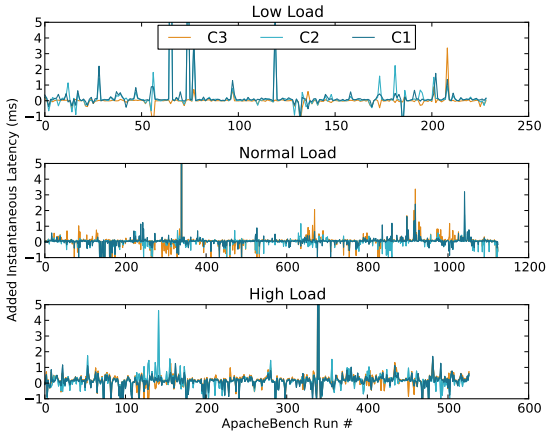


Figure 5.4: The figure shows added latency by aggressively entering different C-states. The latency is given in ms, and normalized per request. Notice that the different workloads contain varying numbers of AB runs, but all have the same 1 hour duration.

Considering the amount of power saved as well as the added latency, directly entering C3 seems to be the most desirable of the policies examined so far.

	Excess completion time			Added variability		
Workload	C1	C2	C3	C1	C2	C3
Low	7.3%	7.4%	0.4%	0.0%	13.0%	38.9%
Normal	0.7%	0.4%	0.5%	0.0%	27.6%	29.2%
High	0.7%	−0.2%	−0.4%	0.0%	12.4%	47.3%

Table 5.3: Summary of performance degradation due to entering C-states directly. Percentages are calculated using means and standard deviations, and are relative to using no PM.

5.2.2 Static Timeout Based C-state Entry

Next, we consider entering a C-state following a period of inactivity. In the light of our findings in the previous section, we limit our experiments to only consider entering the C3 power state on timeouts, rather than examine all possible permutations. We perform the same measurements as in the previous section, and do this for timeout values from $1\mu s$–$1000\mu s$. This choice of values is motivated by real data center (DC) workloads seldom containing long idle periods, but rather very many and short ones [59]. If any energy is to be saved, the timeouts must be relatively short.

The power consumption of entering C3 after a static timeout is shown in figure 5.5. All timeouts exhibit considerable power savings when compared to using no PM. Although the standard deviations are prohibitively large to say anything with certainty, the power consumption seems to increase slightly with the timeout duration. As explained in section 2.2, this should be expected as the amount of potential sleep time that is wasted increases with the timeout. The slight decrease for a timeout of $1000\mu s$ could be attributable to the number of inappropriate state transitions (also explained in section 2.2) decreasing.

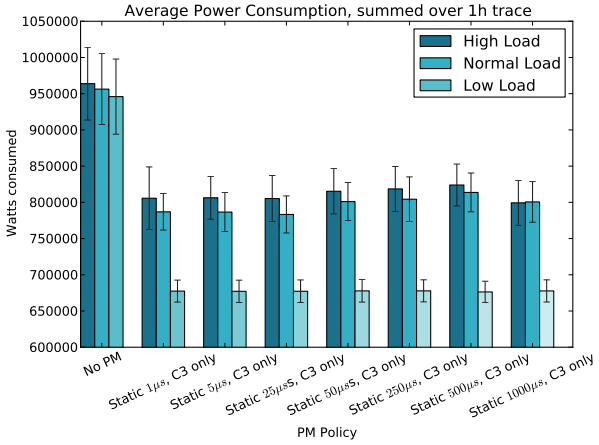


Figure 5.5: Mean power consumed over 1 hour long AB web traffic traces. The amount of power consumed is summed over the entire trace to better show differences between the different timeout values.

We next go on to consider excess completion time resulting from the use of different timeout values. Figure 5.6 shows the mean excess latency over the 1 hour AB traces. For all timeout values, the variability is significantly lower than when using no pm, meaning that timeout-based policies are consistently slower. As can be seen, the excess completion time increases dramatically with increasing timeout values. These results were surprising. Several works mention increasing timeout values as a means to *reduce* the extra latency generated by a PM policy [54], [39]. Recall from section 4.4.2 that with our implementation of entering C-states following a static timeout, a timer is created before the core is halted. The only possible ways for the core to leave this halted state are:

1. The arrival of work, meaning that the pending timer should be canceled.

2. The timer fires, meaning that a sufficiently long period of inactivity has occurred, and a deeper C-state (C3) should be entered.

Measuring the ratio of canceled timers for our different timeout values, it is apparent that as the timeout value increases, so does the number of canceled timers. This is illustrated in figure 5.7.

Another key point is that when the timeout value increases, more and more energy is wasted waiting for the timeout to expire, i.e., the benefit of the PM policy diminishes. At the same time, we reap more and more of the negative effects. If the timeout value is short, the cost of handling the timer is hidden from the users because the system is inactive. The cost of canceling a timer is not hidden, as the core has already been awoken by the arrival of work. Thus, as the number if canceled timers increase, so does the latency experienced by the users.

In [59], the authors claim that adding delays smaller than 1ms when exiting the idle loop does not result in significant impacts on response times. As can be seen in the plots of figure 4.6, the added latency of our static timeout implementation never approaches 1 ms even in the worst case, and on average, adds only 10^3 to 10^4 cycles, or about $0.5\mu s$ - $5\mu s$ if the CPU is run at 2GHz. Non the less, we observe severe effects on latency for even small increases in delay. We thus find our results to be in conflict with the claims of [59].

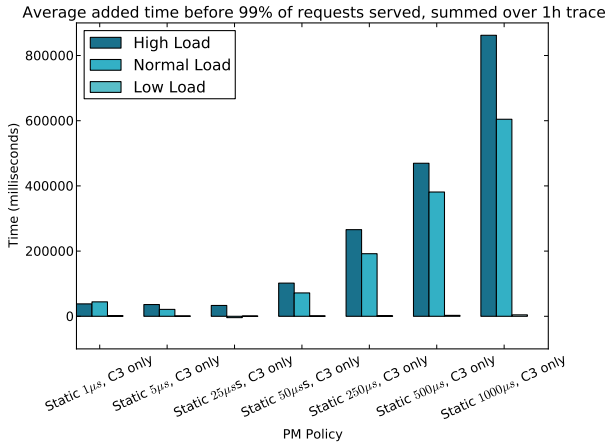


Figure 5.6: Mean excess completion time from entering C3 following a period of inactivity. The measure of completion time is the time before 99% of requests are served, and the plots show the difference between entering C3 after a timeout, and using no PM.

Finally, we turn to the instantaneous user-perceived latencies. For the sake of visualization, we plot only the latency for entering C3 after timeouts of $5\mu s$, $500\mu s$, and $1000\mu s$. As can be seen in figure 5.8, longer timeouts incur considerable extra latency. This is contrary to results from [59], where the authors claim that delays less than 1ms when exiting the idle loop are of little

	Excess Completion Time						
Workload	$1\mu s$	$5\mu s$	$25\mu s$	$50\mu s$	$250\mu s$	$500\mu s$	$1000\mu s$
Low	5.8%	2.9%	2.6%	3.6%	4.9%	9.0%	14.0%
Normal	2.3%	1.1%	0.2%	3.8%	10.1%	20.1%	31.9%
High	2.0%	1.9%	1.7%	5.2%	13.7%	24.1%	44.3%

Table 5.4: Summary of performance degradation due to entering C-state following a static timeout. Percentages are calculated from means, and are relative to using no PM.

consequence. We conclude that when attempting to save power by putting CPU-cores to sleep under normal data center loads using static timeouts, shorter timeout values are preferable.

5.2.3 Select Best Performing C-state

The last C-state management policy we consider, is that of selecting the best performing C-state at runtime. As mentioned in section 4.3, the Share Algorithm is tunable via several parameters. Namely

- $c_{\textbf{latency}}$**:** The weighting of added latency in the cost function. A high value for c_{latency} means that latency will be avoided.
- $c_{\textbf{wasting}}$**:** The weighting of "wasted" energy. A high value for c_{wasting} will result in deeper C-states being entered.
- α**:** The share parameter, which controls how fast a poorly performing expert is able to recover its weight when it starts performing well.
- η**:** The learning rate, which controls the rate of which the weights of poorly performing experts are reduced.

We want our solution to react quickly to spiky workloads, and thus set a high learning rate of $\eta = 25$, while α is kept low with a value of 0.001. The Share Algorithm is robust with regards to the η and α parameters, so in the case that our choice is non-optimal, this will have only a limited effect [39]. We test our solution with constant η and α, but vary the relative weighting of excess latency and wasted energy in three different configurations. The first of these weigh excess latency double that of wasted energy (see table 5.5). The second have equal weights for the two, while the third weigh wasted energy double that of excess latency. Figure 5.9 contains plots showing the

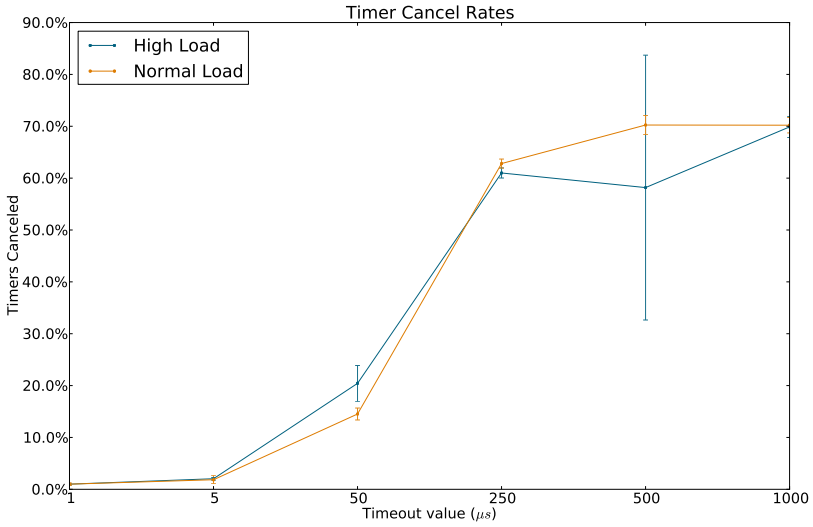


Figure 5.7: Ratio of posted timers that are canceled before firing. Note that this is not a problem for the lowest AB load, and it is thus not included. Error bars show standard deviations.

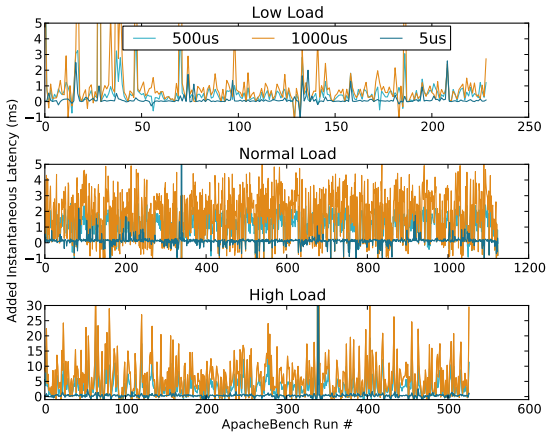


Figure 5.8: Mean user-perceived latency added by using static timeout policies. The latency is given in ms, and normalized per request. Notice that the different workloads contains varying numbers of AB runs, but all have the same one hour duration.

power consumption, completion time, and instantaneous latencies for the three configurations.

	c_{latency}	c_{wasting}
A	2	1
B	1	1
A	1	2

Table 5.5: Description of Share Master configurations. Configuration A corresponds to weighing latency and wasted power 2:1. Configuration B weighs the two equally, while C has a weighting ratio of 1:2.

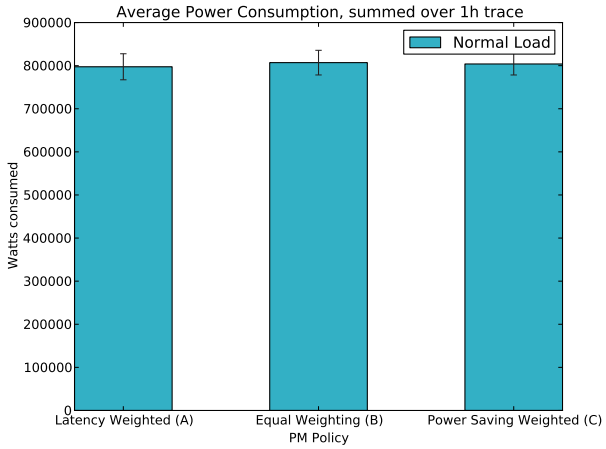
As illustrated by the plots, and numbers in table 5.6, the differences in power consumption are small. However, when we turn to latency it becomes clear that configuration A, which weighs added latency double that of wasted energy, results in significantly lower total completion times. Likewise, the instantaneous latency is lowest for configuration A. Based on these observations, configuration A is used in the rest of the experiments.

	Power Savings			Excess Completion Time		
Workload	A	B	C	A	B	C
Normal	16.6%	15.6%	15.9%	2.0%	5.1%	4.2%

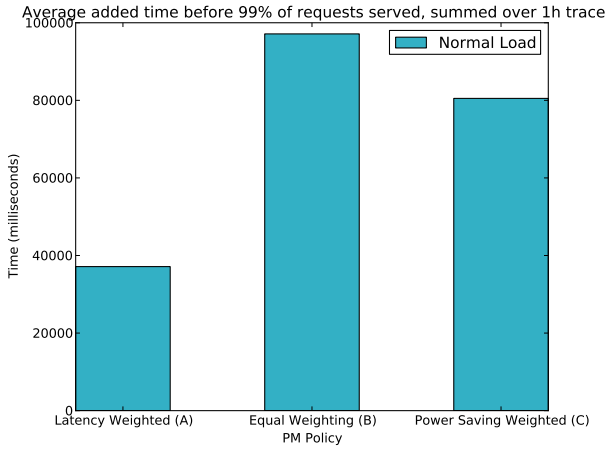
Table 5.6: Summary of Share Algorithm performance with different configurations. Configuration A corresponds to weighing latency and wasted power 2:1. Configuration B weighs the two equally, while C has a weighting ratio of 1:2. Percentages are calculated from means, and are relative to using no PM.

5.2.4 Comparison of C-state Management Polices

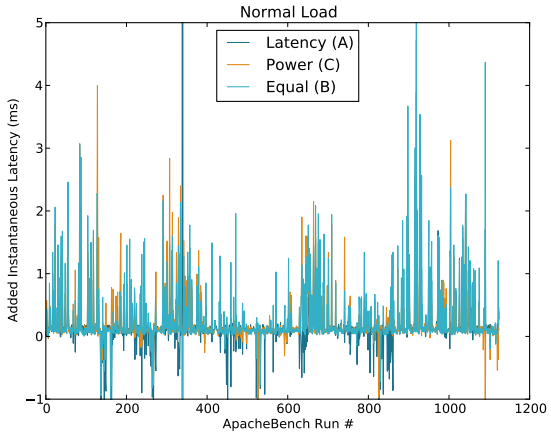
For comparison, we plot the total average power consumption of the best performing C-state policies together in figure 5.10. The figure shows clearly that any of the policies implemented in ROPE will reduce the power consumption significantly. Also, it appears that entering C3 directly is the best solution, as this policy results in the lowest average power consumption over all the different workload traces. This can be seen more clearly in table 5.7, which lists mean relative reduction (when compared to using no PM) in power consumption, and the amount of average excess completion time for the same policies as plotted in figure 5.10. The Share Algorithm selection of the best performing C-state, on average, performs the worst. This is true both with regards to power consumption, and excess completion time. Also, entering C3 directly conserves almost as much energy as the best performing static



(a) Power consumption



(b) Excess completion time



(c) Instantaneous latency

Figure 5.9: Comparisons of Share Algorithm Configurations. The three different corresponds to weighing latency and wasted energy 2:1, 1:1, and 2:1.

policy for both low and high load AB traces (difference is 0.2%), and the most for the high load trace. When looking at the excess total calculation time, it is clear that entering C3 directly performs the best on average.

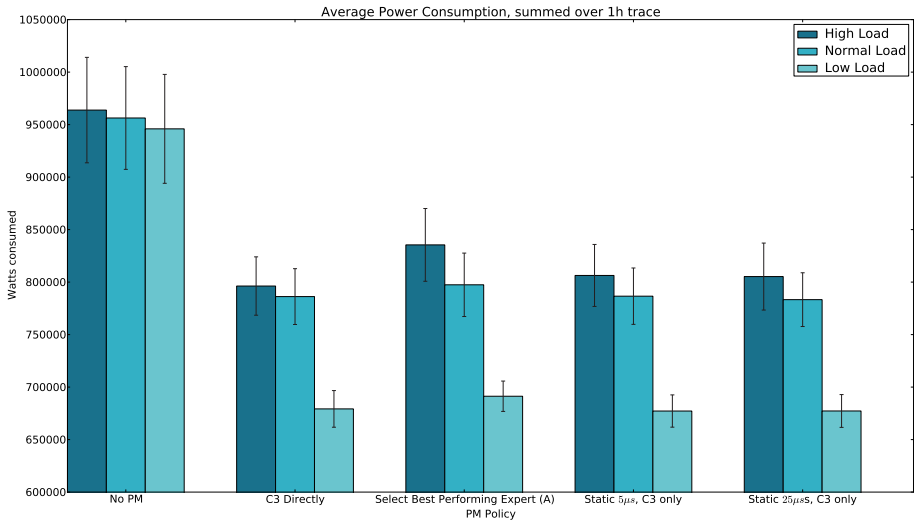


Figure 5.10: Mean power consumed over 1 hour long AB web traffic traces. The amount of power consumed is summed over the entire trace to better show differences between the different PM policies.

When looking at the instantaneous per-request latencies (see figure 5.11), it it is clear that entering C3 directly results in the lowest additional latencies. Critically, in several parts of the trace, using both the Share Algorithm and the static timeout policy with $5\mu s$ timeout result in latency spikes, while entering C3 directly, does not.

We conclude this section by noting that entering C3 directly is the C-state management policy best fit for our workload traces. It proves to be the least intrusive w.r.t latency, and is one of the top candidate policies for all workloads when power savings are considered. In addition, it is both conceptually and implementation-wise the simplest of the C-state management policies in ROPE, and thus fits with our design principles. As a result, we chose entering C3 directly as the default C-state management policy.

	Power Savings			
Workload	C3 Directly	Select Best (A)	Static, $5\mu s$	Static, $25\mu s$
Low	28.2%	26.9%	28.4%	28.4%
Normal	17.8%	16.6%	17.7%	18.0%
High	17.4%	13.3%	16.3%	16.5%

	Excess Completion Time			
Workload	C3 Directly	Select Best (A)	Static, $5\mu s$	Static, $25\mu s$
Low	0.4%	1.3%	2.9%	2.6%
Normal	0.5%	2.0%	1.1%	0.2%
High	−0.4%	3.5%	1.9%	1.7%

Table 5.7: Comparison of C-state management policies. All percentages are calculated from means, and are relative to using no PM.

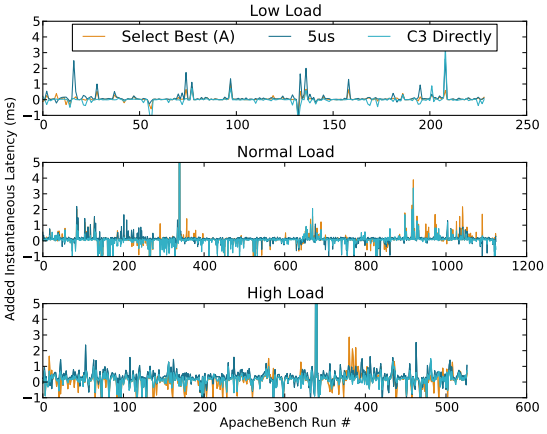


Figure 5.11: Comparison of user experienced latency when employing different CPU C-state management policies. The latency is given in ms, and normalized per request. Notice that the different workloads contain varying numbers of AB runs, but all have the same one hour duration.

5.3 CPU Performance Management

As with C-states, we measure our P-state management polices with regards to power consumption, total completion time, and instantaneous latency. We compare these to each other to determine which solution performs the best. When estimating the prediction quality of our solutions, we use the workload defined in section 5.1.4.

5.3.1 GPHT Predictor

We start by evaluating the implementation-specific properties of our GPHT predictor, and do this using internal measures such as the *prediction accuracy*, and Pattern History Table (PHT) hitrate. We define these internal measures in the following way:

- **Prediction accuracy:** The percentage of predictions made by the GPHT predictor that proved to be correct.
- **PHT hitrate:** The number of lookup operations that resulted in a hit, divided by the total number of lookup operations, i.e. : $\frac{\text{Lookups}_{\text{Hit}}}{\text{Lookups}_{\text{Total}}}$

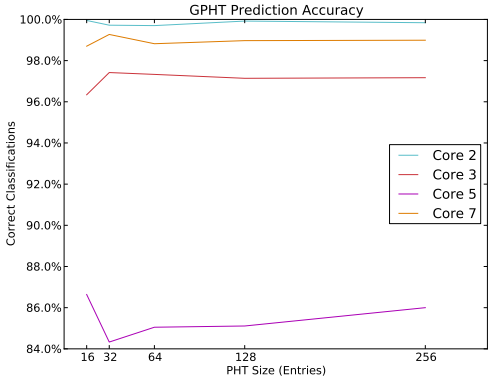
We measure both the accuracy and hitrate for different cores where we generate a synthetic load as described in section 5.1.4.

In [45], and [94], the authors study the properties of a GPHT predictor for varying PHT-sizes. We verify our implementation by measuring the hitrate and accuracy for varying PHT-sizes. Our results are shown in figure 5.12, and corroborate the findings in [94], that accuracy increases with the hitrate. This is especially visible for core 5. We also see that the accuracy increases only slightly with PHT-size, which matches the findings in [45].

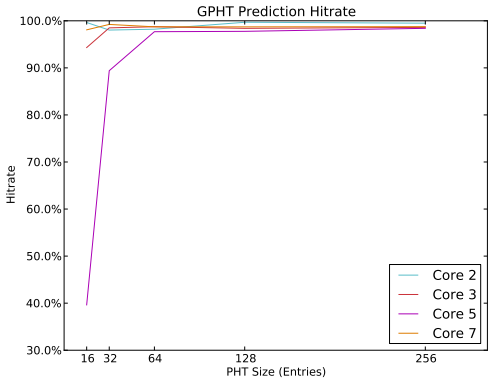
The above results indicate an tradeoff: increase the number of PHT entries to achieve higher accuracy, or keep the PHT-size small to avoid using unnecessary memory. We select a PHT-size of 128 entries, as it seems a fair tradeoff. Throughout the rest of the experiments, we use this size, and a Global Phase History Register (GPHR)-depth of 8.

5.3.2 Comparison of Prediction Accuracy

Next, we compare the naive forecaster to the GPHT predictor described in the previous section. As shown in table 5.8, the naive forecaster consistently



(a) Accuracy



(b) Hitrate

Figure 5.12: Evaluation of GPHT accuracy and hitrate. Both plots created using synthetic workloads similar to those listed in table 5.8, and plotted in figure 4.16.

outperforms the more complex GPHT predictor. We find this result surprising, and investigate the issue further in the following paragraphs.

Many different accuracy metrics have been proposed for evaluating algorithms that produce forecasts based on historical values. However, several of the commonly used metrics are ill-suited for use with real data, as pointed out by Hyndman and Koehler in [44]. The reason why many common measures are unfit for evaluating accuracy in real situations, is that occurrences of zero- and small values result in division by zero problems and very large values. Some examples of commonly used measures that exhibit problematic behavior for real data are:

Load	Accuracy	
	Naive Forecast	GPHT Predictor
Low synthetic	39.6%	36.6%
High synthetic (bursty)	85.8%	84.7%
AB High load	99.9%	99.9%
High Synthetic (stable)	99.6%	99.5%

Table 5.8: Prediction accuracies of P-state prediction algorithms. We find it surprising that the naive forecaster performs so well.

- Mean Squared Error/Root Mean Squared Error - sensitivity to outliers.
- Measure based on percentages - infinite or undefined for occurrences of zero, extremely skewed distributions when observed value is close to zero.
- Measures based on relative error - cannot be used across data series.

In stead the authors of [44] suggest the use of mean absolute scaled error (MASE), which is suitable for all situations, also real data where zero, negative, and very small values occur. The mean absolute scaled error is given by equation 5.1.

$$\text{MASE} = \frac{1}{n}\sum_{t=1}^{n}\left(\frac{|e_t|}{\frac{1}{n-1}\sum_{i=2}^{n}|Y_i - Y_{i-1}|}\right) = \frac{\sum_{t=1}^{n}|e_t|}{\frac{n}{n-1}\sum_{i=2}^{n}|Y_i - Y_{i-1}|} \tag{5.1}$$

Where Y_i is the observed value at time i, and e_t is the error made by the forecaster being evaluated, compared to the forecast given by the naive forecaster at the time t.

When MASE< 1, the algorithm being evaluated results in, on average, smaller errors than the naive forecaster, i.e. predicting the next interval to be equal to the previous one. If MASE> 1, no improvement over the naive forecast is achieved. The only circumstances for which the MASE can be either undefined or infinite, is when all the historical data points being used in the estimation are equal[7]. Other key properties are the ease at which results can be compared, and that it is applicable across data series with different scales. Figure 5.13 shows a plot of the MASE values of different prediction algorithms for our

7. We use this result when deciding on benchmarks for evaluating predictor accuracy.

prediction trace. Note that only the data from a single core was used to generate this plot.

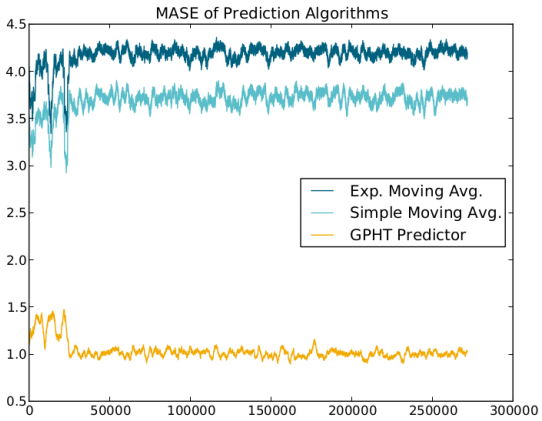


Figure 5.13: The figure shows the MASE of various prediction algorithms. Notice that this plot only contains data from one core, which runs a compute intensive and bursty synthetic load. The MASE is calculated with $n = 2000$.

Using logs collected from runs with our workload for prediction quality testing, we also evaluate the merits of a moving- and exponential moving average predictors offline. These are common statistical prediction algorithms, and their definitions are given in equation 5.2 and 5.3 respectively.

The simple moving average (SMA) is given by:

$$SMA = \frac{p_m + p_{m-1} + ... + p_{m-(n-1)}}{n} \tag{5.2}$$

where n is the number of data points to consider. While exponential moving average (EMA) is defined recursively as:

$$S_1 = Y_1 \text{ for } t > 1,\ S_t = \alpha Y_{t-1} + (1 - \alpha) S_{t-1} \tag{5.3}$$

Where α represents the degree of weighting decrease ($0 < \alpha < 1$), Y_t is the value at time t, and S_t is the EMA at time t.

For all our experiments, we use window length equal to the GPHR in our GPHT predictor, i.e. $n = 8$. For exponential moving average, we use $\alpha = 0.1$. Results for cores running different loads are given in table 5.9

The table shows that the naive forecaster performs the best w.r.t accuracy in three out of four cases. The only workload where any of the statistical

	Naive Forecast	GPHT Predictor		Moving Avg.		Exp. Moving Avg.	
Load	Accuracy	Accuracy	MASE	Accuracy	MASE	Accuracy	MASE
Low synthetic (Bursty)	39.6%	36.6%	1.05	46.8%	0.86	51.0%	0.70
High synthetic (Bursty)	85.8%	84.7%	1.03	58.0%	3.70	51.0%	4.17
AB High load	99.9%	99.9%	Undefined	99.9%	Undefined	99.9%	Undefined
High Synthetic (Stable)	99.6%	99.5%	Undefined	99.5%	Undefined	99.5%	Undefined

Table 5.9: Prediction accuracies and MASE values for P-state prediction algorithms. Notice that MASE is undefined only if all historical data points are equal. This means that the AB trace is unfit for testing predictors, at it represents the trivial case. MASE calculated with $n = 1000$.

approaches seems to have any merit, is the low utilization synthetic load, which corresponds to very short bursts of prime number generation, and short but frequent sleep-times (1ms and 1ms, respectively).

5.3.3 Comparison of P-state Management Polices

We now turn to the external measures. As shown in figure 5.14, there is little difference in the power consumption when using the different P-state management policies. However, the GPHT predictor performs better when we consider the total completion time. As shown by figure 5.14, the naive forecast results in considerably larger completion times. The same is true for the instantaneous per-request latencies, as shown in figure 5.16. Table 5.10 sums these observations.

	Power Savings		Excess Completion Time	
Workload	Naive Forecast	GPHT	Naive Forecast	GPHT
Low	15.9%	15.9%	9.0%	0.6%
Normal	15.1%	13.4%	10.0%	0.5%
High	15.4%	15.2%	7.5%	−1.7%

Table 5.10: Comparison of P-state management policies. All percentages calculated from means, and are relative to using no PM.

Although the GPHT predictor performed worse than the naive forecaster accuracy-wise, it results in much better latencies. Recall that the naive forecaster has substantially lower overhead, 3752 compared to 15456 cycles on average. It appears that the circumstances under which predictions are correct, is more important than the average accuracy, i.e., if a prediction is wrong when the load is low, this is less detrimental to performance than if the wrong prediction is made under high load. We speculate that the GPHT predictor is able to react to patterns in the workload, reducing the latency, perhaps at the expense of wrong predictions during periods of lower load. This could explain

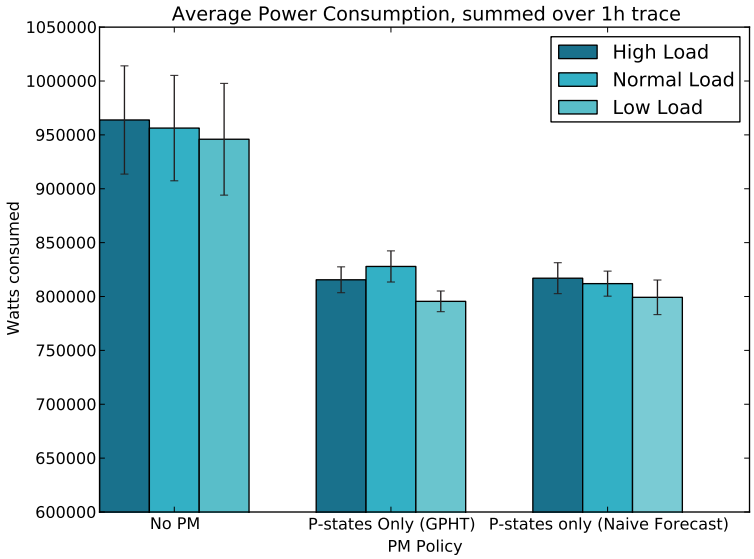


Figure 5.14: Power consumed over 1 hour long AB web traffic traces. The amount of power consumed is summed over the entire trace to better show differences between the use of only P-states and no PM.

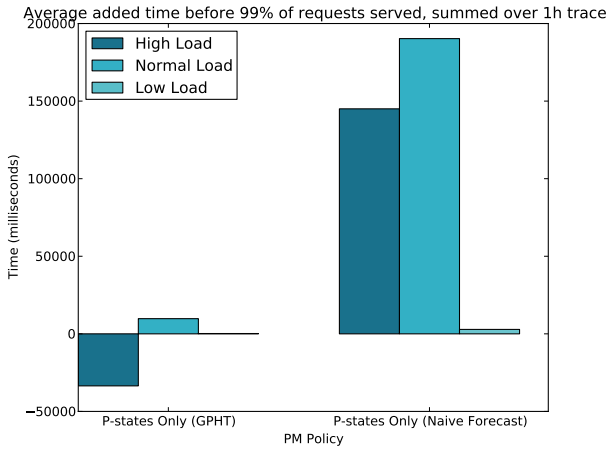


Figure 5.15: Excess latency from using available P-states to save energy. The measure of latency is time before 99% of requests are served, and the plots show the difference between using P-states, and no PM.

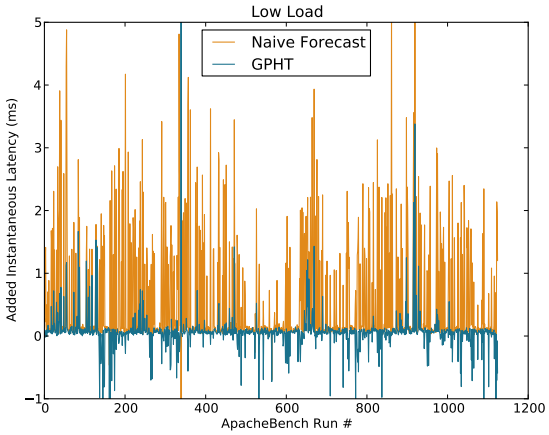


Figure 5.16: Instantaneous latency of using P-states for power management.

the GPHT predictor using slightly more power than the naive forecaster.

To conclude, there is little difference in the energy savings between the naive forecaster and the GPHT predictor. However, we select the GPHT predictor as the default P-state management policy in ROPE, as it results in significantly lower latencies and total completion times for our workload traces.

5.4 Core Parking Scheduler

Finally, we turn to evaluate our energy efficient dynamic round-robin (DRR) scheduler. Recall that the main goal of running our energy efficient scheduling algorithm, is maximizing the number of cores that are sleeping. DRR accomplishes this by packing tasks on a subset of the cores until a threshold, θ_{recruit} is reached for the active cores. Further, cores whose utilization is below θ_{retire}, are moved into the retiring state, where they will remain for a given time, T_{retire}, before ending up as inactive cores. If a new task arrives, it will be assigned to an already active core if this is possible. If the total utilization is greater than θ_{recruit}, a core will:

1. Be unretired, if any cores are currently in the retiring state.

2. Be reactivated, and awoken from the inactive state.

Our implementation of an energy efficient DRR scheduler only considers threads. This is important, as the omni-kernel architecture (OKA) is based on

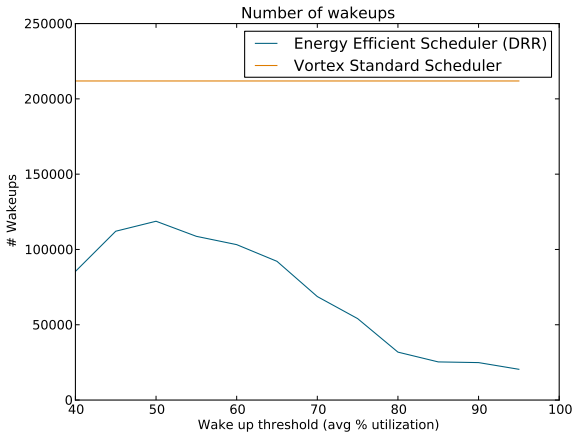
communication through message passing. Because our scheduler is not used for this purpose, the AB workload traces are unfit to test how well our scheduler packs work onto cores. This is because only a tiny fraction of the total workload (the Apache user-level threads) is under the schedulers control. We have experimented with using the DRR scheduler for message processing and other Omni-kernel resource as well, but currently this branch of Vortex is not stable enough to support our long-running experiments. Conceptually, however, the OKA is designed with the idea that plug-in-schedulers, for instance energy efficient ones, can be used for all resources.

Instead of the AB workload traces, we use a synthetic trace as described in section 5.1.6 when evaluating the properties of our DRR implementation. Note however, that we *do* use the AB traces to evaluate the intrusiveness of the scheduler.

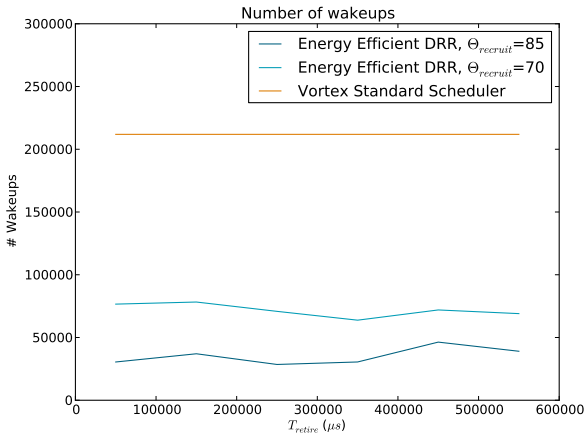
5.4.1 Internal Measures

We use the number of wakeups, i.e. the number of times a core is awoken from a C-state, C_n, $n > 0$, as an internal measure for how well our DRR scheduler is able to pack tasks onto cores. If the number of wakeups is low, this means that many cores are idle, and that the active cores are used to execute arriving tasks. We run a synthetic workload where we fork of a new thread consuming 2% of a CPU-core's compute resources every two seconds. We do this until we reach a total system utilization of 300%, at which point we read how many wakeups have occurred.

We vary θ_{retire} and T_{retire}, and as shown in figure 5.17 a), the number of wakeups is highly dependent on θ_{retire}. As θ_{retire} increases, the number of wakeups decreases. Also visible in a) is a tendency of diminishing returns. At $\theta_{\text{retire}} > 80$, only slight decreases are obtained. However, the latency of using the DRR scheduler will continue to increase, as more and more threads are queued for execution on the subset of active cores. How long threads are queued will depend both on the properties of the thread population, and the configured maximum time slice. For example, if 10 threads are queued, and each of these use their 5ms time slice fully, this would result in $10 \cdot 5ms = 50ms$ of queuing time. Plot b) shows that the number of wakeups is less dependent on the retirement time threshold, and there does not seem to be any general pattern in how T_{retire} alters the number of wakeups.



(a) Varying θ_{recruit}, constant $T_{\text{retire}} = 500000$ and $\theta_{\text{retire}} = 50$.



(b) Varying T_{retire}, constant $\theta_{\text{recruit}} = 85$ and $\theta_{\text{retire}} = 50$.

Figure 5.17: Properties of energy efficient DRR scheduler. All values are means calculated for at least 15 measurements over four different physical hosts.

5.4.2 Comparison with Standard Vortex Scheduler

In this section we compare our DRR implementation to the default thread scheduler in Vortex. This scheduler works by assigning a thread to a core in a round-robin fashion the first time it is run. The thread will then continue to run on this core until is is finished or destroyed.

Recall that the energy efficient DRR scheduler is intended to compliment existing C- and P-state management policies. When testing our scheduler, we use the following common configuration:

- **C-state policy:** Enter C3 directly.
- **P-state policy:** GPHT predictor with GPHR depth=8, and PHT-size=128.

We run three different workloads, each created by forking of a new thread consuming 2% of a CPU-core's compute resources every two seconds. This is done for varying global utilizations of 150%, 350%, and 500%.

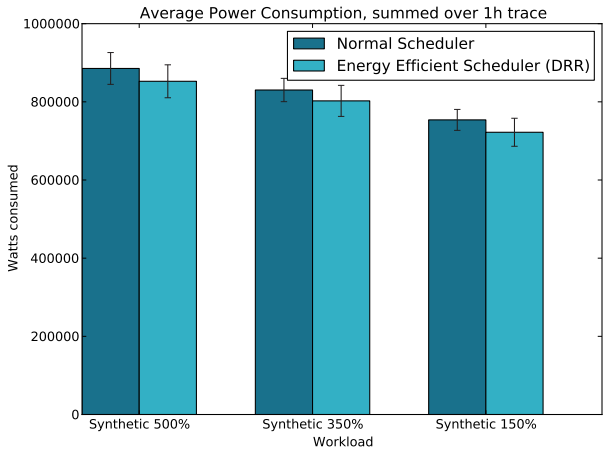


Figure 5.18: Power consumption of synthetic loads using energy efficient DRR scheduler.

Figure 5.18 shows the power consumption with the default and DRR scheduler. As can be seen, the energy efficient DRR scheduler consumes less power on average for all three workloads. Table 5.11 lists these differences in percentages: relative to using the default scheduler, energy efficient DRR consumes 3.5%–4.2% less power on average.

Workload	Relative Power Saving
Low	4.2%
Normal	3.5%
High	3.7%

Table 5.11: Relative energy saving when using energy efficient scheduler. Percentages are calculated from means, and are relative to using the Vortex standard scheduler.

5.4.3 Effects of Energy Efficient Dynamic Round-Robin Scheduling

We now describe the effects of introducing the energy efficient DRR to Vortex. Figure 5.19 shows the amount of power consumed over the one hour long AB traces for three combinations of PM policies that are facilitated by ROPE. Also plotted, is the power consumption if no PM is performed. As expected, when serving AB workload traces, there is no decrease in power consumption when introducing energy efficient DRR scheduling. As mentioned, this is because most of the workload is outside the schedulers control.

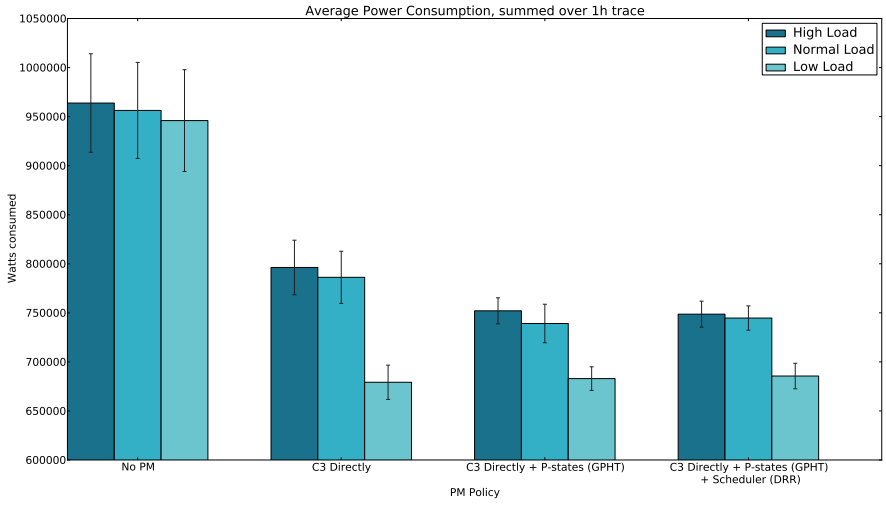


Figure 5.19: Mean power consumed over 1 hour long AB web traffic traces. The amount of power consumed is summed over the entire trace to better show differences between the different PM policies.

With regards to total completion time, energy efficient DRR does not seem to have a significant impact when compared to using only a combination of

entering C3 directly and the GPHT P-state predictor. A plot of this is given in figure 5.20. The same is true for instantaneous latency, which is plotted in figure 5.21.

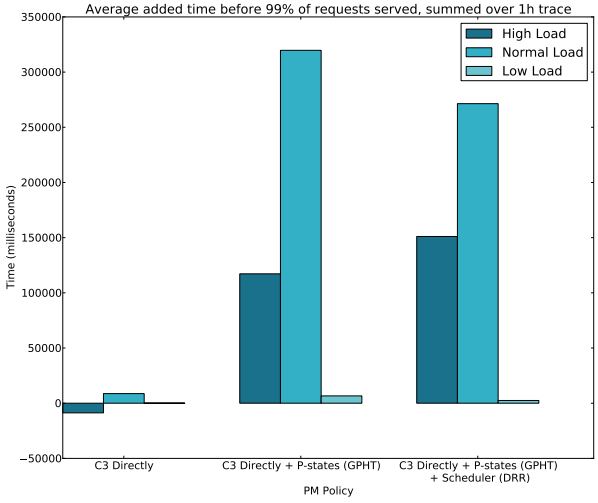


Figure 5.20: Mean excess completion time from using different combinations of PM policies. The measure of completion time is the time before 99% of requests are served.

Table 5.12 shows the relative power savings and added completion times for the three combinations of PM policies. The numbers do not indicate any detrimental effect of introducing energy efficient DRR when serving AB workload traces. Also, as shown in table, 5.11 energy can indeed be saved by the scheduler. When interpreting these numbers it is important to notice that AB is a closed-loop benchmark. This means that if AB is run with a concurrency level of n, one connection must finish before the next is initiated. This means that very small delays will impact the throughput directly, and can thus sum to relatively large numbers. In effect, this means that the measure of excess completion time might exaggerate the actual performance implications to some extent.

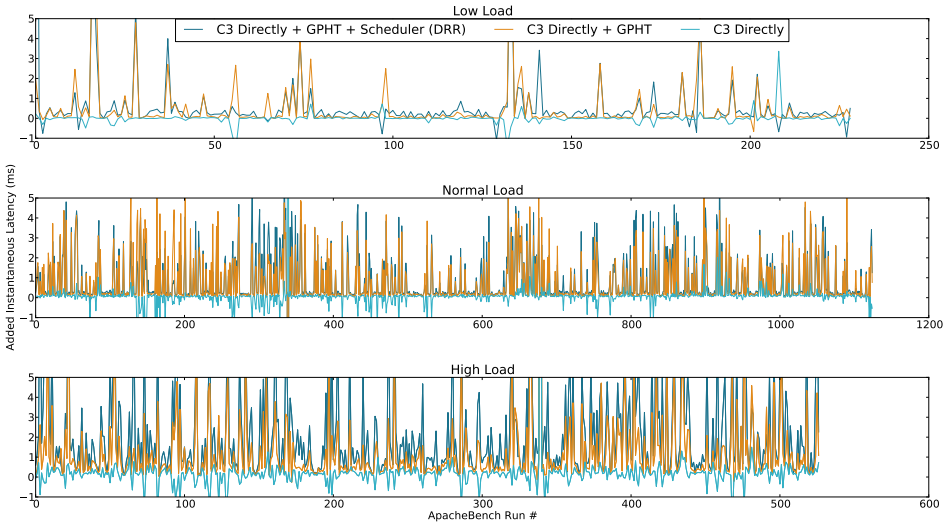


Figure 5.21: Mean user-perceived latency added by using various PM policies. The latency is given in ms, and normalized per request. Notice that the different workloads contain varying numbers of AB runs, but all have the same one hour duration.

	Power Savings		
Workload	C3 Directly	C3 Directly + GPHT Predictor	C3 Directly + GPHT + DRR
Low	28.2%	27.8%	27.5%
Normal	17.8%	22.7%	22.1%
High	17.4%	22.0%	22.3%

	Excess Completion Time		
Workload	C3 Directly	C3 Directly + GPHT Predictor	C3 Directly + GPHT + DRR
Low	0.4%	20.9%	7.8%
Normal	0.5%	16.9%	14.3%
High	−0.4%	6.0%	7.8%

Table 5.12: Comparison of selected PM policies in ROPE. All percentages calculated from means, and are relative to using no PM.

5.5 Performance Comparison of Power Management Policies

While we use AB traces to measure power consumption and performance in terms of latency, we turn to TPC-C and HammerDB for measuring throughput. As described in section 5.1.5, TPC-C is a benchmark for relational databases that mimics the usage patterns of operators making queries and updates in a series of warehouses.

Table 5.13 contains the results from the TPC-C benchmark run with HammerDB. There are some key points in this table that we want to emphasize:

- Employing no PM results in the highest performance.
- When using only P-states to conserve energy, the naive forecaster performs better than the GPHT predictor. This is to be expected, as both TPM and NOPM are measures of throughput, and the overhead of the naive forecaster is significantly lower than for the GPHT. That is, while the GPHT predictor manages to provide better response times, the naive forecaster gives higher throughput.
- Both of the P-state management policies result in lower levels of performance than the C-state management policies. We believe this is because of the intervals at which we adjust the frequencies of CPU cores, which is every 100ms. By only entering C3 directly, the C-state management policies are able to race-to-halt, finishing of any work as quickly as possible. The P-state policies, on the other hand, risk remaining in at a prohibitively low frequency for the duration of the sampling interval. We think that simply reducing the time between P-state management predictions will alleviate this issue.
- When comparing the C-state management policies to each other, it is difficult to make any claims with regards to performance, but the standard deviations, and thus the variability, seems to be larger when entering C3 directly (see also the two combinations of C- and P-states).
- The introduction of the energy efficient DRR scheduler reduces performance considerably. This is not unexpected, as the scheduler is naive in its nature. No action is taken to keep caches warm, and threads risk being shuffled between cores frequently. Further, because the threshold for recruiting additional cores is relatively high ($\theta_{\text{recruit}} = 85$), threads can experience increased queue times. In addition, the implementation

itself is not optimal. For instance, a global lock is used to ensure mutual exclusion when obtaining scheduling decisions. As system-wide utilization increases, this lock will be prone to contention.

PM-policy	TPM	NOPM
No PM	732615 ± 108647	11208 ± 1660
P-states Only (GPHT)	630601 ± 81778	9133 ± 2376
P-states Only (naive forecast)	653254 ± 69413	9991 ± 1058
C_3 Directly	690642 ± 130245	10554 ± 1982
Select Best Performing	698429 ± 88799	11208 ± 1660
C_3 Directly + P-states (GPHT)	663692 ± 120747	10155 ± 1840
C_3 Directly + P-states (naive forecast)	635293 ± 82986	9723 ± 1273
C_3 Directly + P-states (gpht) and Scheduler	381512 ± 32747	5834 ± 500

Table 5.13: Summary of TPC-C performance for various PM algorithms (mean $\pm$ std. deviation).

5.5.1 Summary

In this chapter we have give detailed descriptions of our experiments and the properties of the PM policies available in ROPE. We have seen that some of our results contradict those presented in previous work. In chapter 7, we will discuss our findings in more detail.

6

Related Work

This chapter describes related work. Power management of computer systems is a vast field, and we limit our review to works regarding power management (PM) of CPUs, and energy efficient scheduling. We also discuss some of the other approaches to tackling the issues of energy efficiency in cloud computing and data centers (DCs).

6.1 Power Management of CPUs

We start by discussing a selection of previous work focusing on PM of CPUs.

When adjusting CPU core frequencies, we use the "mem/μ-ops" ratio to classify workloads into different execution phases similarly to Isci et al. [45]. In their work, the authors perform statistical machine learning by employing a GPHT predictor. This predictor is instantiated within Linux as a loadable kernel module (LKM), and is used to make DVS decisions. While we focus on multi-core x86, Isci et al. target a uni-core mobile platform. Further, the entire operation of the solution proposed Isci et al. takes place within a performance monitoring interrupt (PMI) handler.

Likewise, Moeng and Melhem [61] use statistical machine learning to adjust CPU frequency and voltage. Using a decision tree, their machine learning algorithm maximizes energy efficiency. Like our solutions, each core indepen-

dently sets its frequency and voltage. Further, cache access- and miss rates obtained from PMCs are used to classify phases of execution. The obtained data is then manipulated and compressed into a decision tree, which is used to decide frequency and voltage at runtime. Unlike our GPHT solution, their algorithm is not trained online. Instead, the system is trained prior to deployment by running different workloads and having cores run with random frequencies and voltages.

A somewhat similar approach is taken by AbouGhazaleh et al. [9]. PMCs are used to sample metrics such as the number of cache accesses and last-level cache (LLC) misses. These are used to split the workload into execution phases, which in turn are used to select frequency and voltage. Similarly to [61], a supervised machine learning technique is used to train the system prior to deployment. Apart from being pre-trained, their solution differs from ours in two distinct ways. First, their solution relies on a power-aware compiler in order to characterize execution phases. Second, they focus on embedded processors with multiple clock domains. In contrast, all policies in ROPE are independent of the applications being executed.

In [74], Rajmani et al. employ a scheme very similar to our GPHT predictor for adjusting performance. A three part solution built on monitoring, prediction, and effectuation is used to select CPU P-states obtained via ACPI tables. Also, PMCs are used to characterize the workload. Their solution differ from ours in that the employed prediction scheme is based on a static set of equations. Thus, it is unable to adapt to the workload over time. Further, their entire solution is controlled by a user-level application that accesses drivers to monitor workload behavior and adjust the CPU frequency. ROPE, on the other hand, resides entirely within the Vortex kernel.

Finally, in [47], Jung and Pedram present a PM framework for multi-processor systems that predicts a performance state, and then extracts an optimal PM decision from a pre-computed policy table. The predictions are made using a Bayesian classifier, which has been pre-trained using supervised learning. The authors main rationale for choosing this solution is that it minimizes overhead related to the classification process. The ROPE GPHT predictor is an online algorithm, and sacrifices latency for the ability to dynamically adapt to any workload.

6.1.1 Share Algorithm

The Share Algorithm has been employed for power management on multiple occasions. In [39], an approach similar to ours is used to perform adaptive disk spin-down for mobile computers. They find the algorithm to be capable

of successfully adapting to the recent disk activity, performing better than other previously known approaches.

In [54], the authors target power management of the CPU by employing the Share Algorithm. They also employ RLE to minimize the memory footprint, but their solution differs from ours in that they only target power management for mobile computers at the CPU-package level. They also use statistical sampling when generating their RLE traces. We focus on power management on a per-core basis, enabling finer granularity in the PM decision making process. Both [39] and [54] use the Share Algorithm to create a weighted sum of expert outputs—each being a static timeout value. Our experiments indicate this solution to be unfit for our problem, as the reasoning behind their cost functions implies naively busy-waiting for timeouts to expire.This would increase power consumption unnecessarily.

A solution much like ours is used in [28] to support dynamic power management of both hard disk drives and a WLAN card. Like in our policy for selecting the best performing C-state, the authors use the Share Algorithm to select a single best expert to use at any point in time. The same approach is also used in [37] to select one among several available caching policies. Dhiman and Rosing use the the Share Algorithm to select the best performing frequency/voltage pair [29]. One expert is used to represent each of these pairs, and PMC readings similar to ours are used when calculating the loss of each of the experts. However, the PMC readings are logged and maintained per-task. This enables the selection of frequency and voltage based on the currently executing task's observed properties.

6.2 Energy Efficient Scheduling

Next, we look at energy efficient scheduling.

In [91], Weissel and Bellosa propose an energy-aware scheduling policy based on the use of event counters. By exploiting the information present in PMCs, the scheduler is able to determine the appropriate DVS setting. This is done on a per-thread basis, and the scheduler scales the CPU voltage each time a thread is scheduled for execution. The chosen frequency is calculated by evaluating the event rates in the recent history of the thread. In contrast, our energy efficient DRR scheduler only considers the utilization of the CPU-cores in order to assign tasks. It is also completely decoupled from the DVS management.

Cai et al. [16] use two techniques; (i) meeting point thread characterization,

and (ii) thread delaying; to reduce energy consumption in parallel applications. The first technique is used to characterize threads as either critical or non-critical. Critical threads—the ones that run slowest—should be run at the maximum available frequency to avoid performance degradation, while DVS can be used to scale down cores executing non-critical threads. In the ideal situation, threads on different cores are executed at the frequencies that allow them to reach their synchronization point simultaneously. The authors extend this in [73], where threads with similar criticality are scheduled onto the same core via thread migration. This increases the chances that threads reach the synchronization point at the same time, allowing even greater power savings. Again, the tight coupling between scheduling and DVS is fundamentally different from what we have implemented in ROPE.

In [93], Ye and Xu propose a machine learning based DPM framework for scheduling tasks on multi-core processors. The scheduler, which is based on reinforcement learning, assigns tasks to cores such that an tradeoff between idle times and performance is achieved. Lin et al. [55] presents a system for energy efficient scheduling of VMs that is similar to our energy efficient topology agnostic DRR. Their VM-scheduler relies on the same structure of active-, retiring-, and inactive hardware. They also use similar mechanisms for determining when to move hosts between the different states. In many ways, our scheduler can be seen as a re-purposing of this work, although the massive difference in timescales (their VMs are load balanced between hosts in the order of minutes and run for many hours) lead us to enforce and choose our thresholds differently. Many of the ideas and concepts used in our energy efficient scheduler are also discussed by Siddha et al. [83] in their proposed power saving Linux scheduler.

A large body of work targets how to best run tasks in time sensitive environments while reducing the power consumption. A DVS capable real-time scheduler is presented by Pillai and Shin in [72]. Their scheduler provides the energy savings of conventional DVS approaches, while preserving deadline guarantees. In [38], Gruian provide hard deadline guarantees by employing both online and offline decision making. Yuan and Nahrstedt presents *GRACE-OS*, an energy efficient soft real-time CPU scheduler in [95]. GRACE-OS integrates DVS in a soft real-time scheduler that controls when, how fast, and how long to execute tasks. Scheduling decisions are made according to probability distributions of cycle demands. These distributions are obtained via online profiling.

6.3 Power Management in Data Centers and Cloud

Much work has gone into designing solutions that intelligently place workloads at different physical hosts within data centers. In [64], Moore et al. leverage information about hot and cold locations within DCs to create temperature aware scheduling algorithms. Their proposed solution places incoming workloads intelligently throughout a DC, and is able to reduce cooling costs with a factor of two when compared to location-agnostic solutions. In [13], Bash and Forman study the effect of placing tasks at cooling efficient locations within a DC. They propose a method for ranking servers according to cooling-efficiency, and experimentally validate that there is substantial potential for energy savings if the cooling characteristics of the DC is taken into consideration.

In [43], the authors explore power-efficient consolidation and distribution of VMs. They claim that maximum power-efficiency can be obtained if the attributes of applications running within VMs are taken into consideration, and that maximum power-efficiency is achieved when VMs are consolidated such that resources available at a physical host machine are fully utilized. They propose a load balancing algorithm that combine jobs that consume different resources on the same physical hosts.

Sharifi et al. propose an energy-aware VM scheduling algorithm in [81]. This algorithm is formulated using a set of objective functions describing a consolidation fitness metric. Their proposed solution minimizes the total energy consumption of physical hosts in a whole DC while only incurring marginal performance degradation.

In [32], Femal and Freeh employ a distributed algorithm to allocate power, as opposed to work, to different physical hosts. Their goal is to increase aggregate performance while distributing the available global power under a set of operating constraints. One such constraint could be the globally available power, as limited by the actual circuits. In this case, the globally available power is allocated to hosts according to their contribution to processing the aggregate workload. This way, maximum throughput can be achieved while respecting upper bounds for power consumption.

Kotla et al. [51] use execution characteristics of the work currently running to predict the achievable performance at available frequency settings. They slow the nodes non-uniformly in response to their performance demands, essentially running each CPU at the lowest possible frequency able to meet performance requirements. Using this approach, they are able to maintain

user perceived performance while responding to fluctuations in the amount of available energy. Similarly, Sharma et al. [82] provide power-aware QoS in DC webservers. Their solution minimizes energy consumption while meeting per-class delay constraints. Using a feedback loop, they regulate CPU frequency and voltage to keep the resource utilization around a schedulability bound. By enforcing this bound, they ensure that deadlines are met, effectively running the tasks as slowly as possible while still preserving QoS.

With *VirtualPower* [66], the authors present a system that integrates PM with virtualization technologies commonly employed in DCs. By exposing "soft" versions of the underlying hardware power states, each guest OS is allowed to implement its own PM policy. These decisions are then globally coordinated by VirtualPower, and the updates made to soft VM power states are mapped to actual power states or allocation of virtualized hardware. Using this scheme, power consumption is minimized while the ability to meet application requirements is retained.

In [34], Goiri et al. present Parasol: a prototype green energy data center, and GreenSwitch: a model-based framework for dynamically scheduling workloads and selecting which energy source to use. Parasol features multiple power sources, namely a set of solar panels, a battery bank, and a standard grid connection. GreenSwitch predicts future workloads and renewable energy production, which in turn are used together with current battery levels to generate energy- and workload schedules by a *solver*. These schedules determine how much energy should be used in the future. Using GreenSwitch and a set of MapReduce workloads the authors demonstrate that intelligent management of energy sources and clever placement of workloads can results in significant reductions in energy costs. Aksanali et al. also leverage predictions about future available green energy in [10]. Their work presents an adaptive DC job scheduler which based on the short term predictions of available wind- and solar energy scale the number of scheduled jobs. In addition to reducing the average task completion time for batch jobs, the consumed amount of non-renewable energy is reduced.

In [58], Mathew et al. propose a technique to power down content delivery network (CDN) servers during periods of low load. Their solution seeks to maximize power savings while minimizing both client perceived performance degradation, and wear and tear on hardware resulting from to excessive on-off server transitions. Their experiments show great prospect: reducing energy consumption by 55% while still maintaining service level agreements (SLAs) and minimizing server wear and tear.

Chihi et al. [18] also scale the number of active VMs in a cloud up and down automatically. This scaling is based on a prediction module built on a neu-

ral network. By auto-scaling the available cloud resources according to the actual workload, server utilization is increased and power consumption is reduced.

In [59], Meisner et al. introduce the PowerNap server architecture. In stead of focusing on traditional and complex solutions for fine-grained control of power- and performance states, load balancing, and so on, PowerNap aims to reduce energy consumption by the use of non-traditional hardware. By responding to instantaneous load, the system rapidly transitions between a normal high-performance state, and a near-zero power idle state. As soon as a server exhausts its work queue, it transitions into the *nap* state, consuming almost no power. When work arrives, processing is started in the nap state and a transition to the high-performance state is initiated instantly. Using simulations with real traces of normal server loads, the authors show results superior to traditional DVS approaches with respect to both energy savings and response times.

Using *Elastic Tandem Machine Instances*, Dürr couples low power system on chip (SoC) machines and high powered VM instances [31]. The low power machine is always on, serving low load and ensuring availability. When load rises, the high performance VM is activated, and the workload is transfered to it using software defined networking techniques. The approach is proved viable using a real prototype.

6.4 Reduction of CO_2 Emissions

With the Stratus system [30], the authors use Amazon EC2 cloud instances to model load balancing between different DCs in order to reduce CO_2 emissions and electricity costs. They leverage the fact that the cost of electricity as well as the greenhouse gas emissions will vary over time and with type of power plant. The authors use graph algorithms to guide routing of requests towards different DCs based on the priorities of the cloud operator.

In [62] Moghaddam et al. utilize intelligent live migration of VMs in Virtual Private Clouds (VPCs) to reduce carbon footprint of cloud operation. They leverage that energy used by different DCs come from different sources, and thus result in varying greenhouse gas emissions. Using a genetic algorithm for consolidating VMs across DCs, they minimize the carbon footprint by placing loads whenever the energy is the “greenest”.

/7

Discussion and Concluding Remarks

In this chapter, we start by discussing some of our findings from chapter 5. We then turn our attention to the omni-kernel architecture (OKA), and describe how we believe PM and energy efficiency can be accommodated within this architecture. Following our discussion, we will list our contributions and achievements. We then conclude the thesis.

7.1 Discussion

7.1.1 Findings

During our experiments and evaluations, we have encountered two situations where our results differ from those presented in previous work. In this section, we will discuss each of these in more detail.

Shallow Power States Are Easier to Use

Benini et al. argue in [14] that shallow power states with low associated entry and exit costs are easier to use and more important than deeper power states whose associated costs are higher. We do not argue the correctness of this

conjecture in general, but rather note that our experiments do not suggest this for our CPUs. Specifically, our experiments indicate that entering C3—the most expensive of our platform's CPU C-states—results in both larger power savings and better performance than using the shallower (and cheaper) C1 and C2 states (see Section 5.2.1 for numbers).

We have been unable to find any work documenting why this is, but observe that Microsoft Windows aggressively enter the deepest available C-state in their core parking functionality [79, Ch. 8]. We doubt that this choice has been made arbitrarily.

One possible reason that entering deep C-states is often avoided, might be that the ACPI specification demands that the OSPM keeps state and flushes caches when states deeper than C2 are entered [22]. However, on modern CPUs, this is no longer necessary as the platform manages such state itself [2]. Also, the effects of flushing caches when entering deep C-states might not be very severe for typical web-based workloads [59].

Latencies Below 1ms Are OK

In [59], the authors claim that excess latencies of duration less than 1ms are of little concern if they occur while exiting the idle function. Our experiments contradict this. We find that when leaving the idle loop, even very short excess latencies (for instance the 1–2μs it takes to cancel a timer) have a significant detrimental effect on both total completion time and latency experienced by clients (see Figure 5.6 and Table 5.4 for numbers).

7.1.2 Power Management in the Omni-kernel Architecture

The OKA provides unprecedented control over resource allocation and consumption. Just as meticulous accounting is applied to CPU cycles, consumed memory, throughput, etc., resource records can be be kept for any metric relevant to PM. By developing models that allow mappings from metrics to consumed energy, the OKA implicitly provides total knowledge of:

1. Where energy has been consumed.

2. Who is responsible for the power consumption.

Schedulers can leverage this information to make informed decisions about how a task should be executed, for instance by adjusting the frequency of CPUs according to the properties of the currently executing thread [91]. Sim-

ilarly, information present in resource consumption records can be used to determine energy efficient schedules by reducing the number of state transitions, e.g. by scheduling tasks with similar frequency needs following each other. By using metrics such as the number of LLC-misses, tasks can be placed by the scheduler so that minimal contention of shared resources occurs.

The OKA is built around the notion of resources, and each of these is governed by a scheduler. As such, different resources can employ energy efficient schedulers tailored specifically to accommodate the needs of the resource. For example, some resources might be I/O bound, while others are compute intensive. In this case, the scheduler governing the I/O bound resource could reduce CPU frequencies, while a scheduler for a computationally expensive resource might increase the frequency—racing to halt. The necessary profiling of the resources could be based on the resource records, and be performed online.

7.2 Contributions and Achievements

In this thesis we have described the design and implementation of ROPE, a system for OSPM in the Vortex OS kernel. We have added ACPI support, and also functionality for Intel-specific PM to the Vortex kernel.

ROPE contains a selection of PM policies for CPUs that use CPU C- and P-states. We have evaluated these policies, and determined which performs the best. Our evaluations have uncovered discrepancies between claims in previous work and our experimental observations. We also implemented a naive energy efficient scheduler. The implemented functionality is modular, and policies can be chosen according to the expected workload.

Through evaluation of our policies, and the scheduler in particular, we have shown that the OKA is dependent on multiple power aware schedulers if maximum energy efficiency is to be achieved. This is due to different resources being governed by different schedulers, and normal workloads rarely being constrained to a single resource.

With ROPE, we have achieved power savings in the order of 17%–28% for realistic web-based workloads. While we have achieved a 17% reduction with no detectable performance degradation, the higher energy savings come at the cost of increased latency. However, we argue that our numbers might exaggerate the actual performance impact.

7.3 Future Work

ROPE has now been deployed in the Vortex kernel, and we are already saving energy in our server racks. However, we would like to continue development of the functionality necessary to do energy efficient scheduling of more than just threads. As was demonstrated by running our AB workloads, the approach of only scheduling threads is futile when the workloads result in high resource consumption inside the kernel.

Another natural step would be to implement a topology-aware scheduler that is able to keep caches warm, and the number of active CPU packages to a minimum. Such a scheduler could also be expanded in the direction of using resource records to monitor energy consumption of tasks and tenants. This could open up for new and interesting approaches to PM in Vortex.

Through the evaluation of PM policies, we have identified that a key aspect of achieving energy efficiency in the OKA is to provide power-aware schedulers for all resources. We particularly wish to look into which resources are CPU/memory bound, and whether batch processing of messages can be used together with scheduler-directed DVS to reduce energy consumption further.

7.4 Concluding Remarks

We have successfully deployed ROPE—a system for OSPM in the Vortex kernel. ROPE has been designed and implemented according to our design principles. We have evaluated all the PM policies available in ROPE and quantified their impact on both power consumption and performance. As such, the user can make informed choices about energy/performance tradeoffs.

Finally, as part of the Vortex kernel, ROPE will continue to be in active use by faculty and students.

A

ACPI Objects and Namespace

This appendix details some of the central ACPI objects used throughout the Vortex PM functionality. Also, a brief introduction to the ACPI namespace is included.

A.1 The ACPI Namespace

ACPI is presented by the platform to the OS as a single hierarchical namespace of *objects* (see figure A.1). These objects allow device detection and configuration, and several classes exist. The contents of objects are varied, but most refer to data variables, control methods, or functions provided by platform BIOS. In the following section, various objects used in the implementation of ROPE are explained.

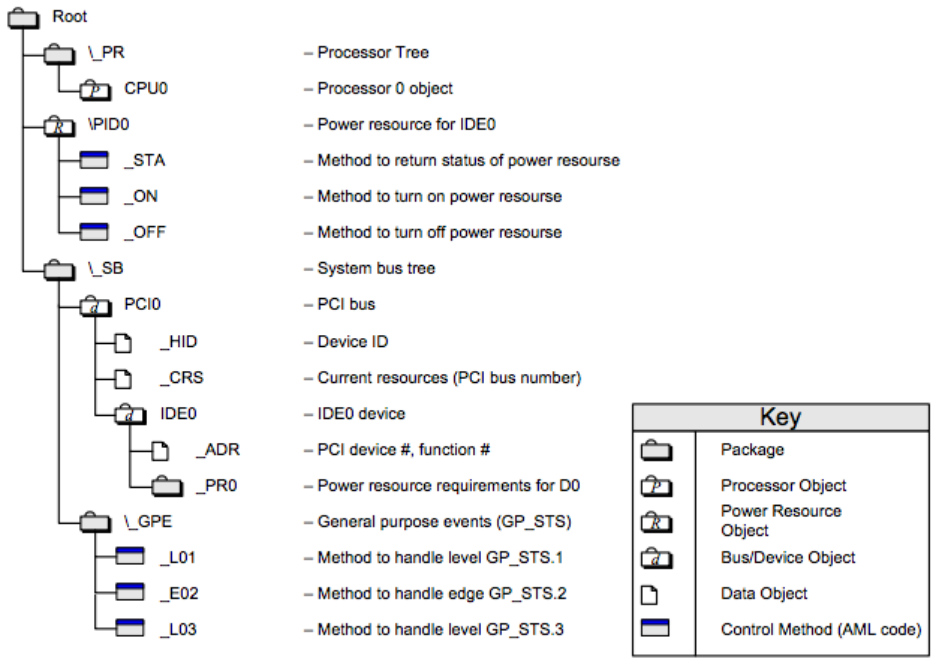


Figure A.1: Example of ACPI namespace. Figure borrowed from the acpi specification v. 5 [22].

A.2 ACPI Objects

A.2.1 The _OSC Object

The operating system capabilities(_OSC) object is an optional control method that can be used by OSPM to notify the platform of support for different features. In the case of processors, such information could be whether OSPM wish to handle various features such as P-state coordination itself or rely on the platform and hardware to handle this for it.

The _OSC object is used as follows:

1. The _OSC object for a device is located in the ACPI-namespace.

2. The arguments are set, including a UUID, and *capabilities buffer* formated specifically to match the device.

3. The method is evaluated, and the status can be verified by the returned values. If the bits in the capabilities buffer were set in such a way that OSPM code indicated the capability of handling any features, the platform may respond by generating new ACPI objects for such features in the namespace.

It should be noted that the _OSC object is used for the same purpose as the _PDC object which it replaced in ACPI version 3.0. A detailed description of the _OSC method and its usage can be found in [22, p. 282–291]. For details regarding the format of the capabilities buffer, device specific documentation must be consulted.

A.2.2 The _PSS Object

The performance supported states(_PSS) object is an optional object indicating the number of supported processor performance states. When evaluated, this object returns a list containing information about available performance states including the internal CPU core frequency, power dissipation, control register values needed to transition into performance states, and status register values enabling the OSPM code to verify that transitions were successful. An example of an ACPI-package returned by the _PSS object is illustrated in figure A.2.

A.2.3 The _PPC Object

The performance present capabilities(_PPC) object is a dynamic method used by the OSPM code to obtain information about which performance states that can be used at any given point in time. When evaluated, the object returns a single integer describing the highest(lowest numbered) performance state that is currently available. For instance, if the value 1 is returned $P0$ is unavailable, while $P1$ through Pn are available.

To perform a transition, the _PPC object must first be used to obtain information regarding which P-states can be entered. The OSPM software can then choose any available performance state, and obtain the information necessary to transition into it from the _PSS and _PCT objects. That is, which control and status values to be read and written to which registers. This process is illustrated in figure A.3.

A.2.4 The _PCT Object

The performance control(_PCT) object is an optional object that can be used by OSPM to transition a processor into a given P-state. Evaluation of this object returns an ACPI-package containing a *control-* and *status register*.

The OSPM software can enter a desired a performance state by writing a value specific to said state into a performance control register. When doing this,

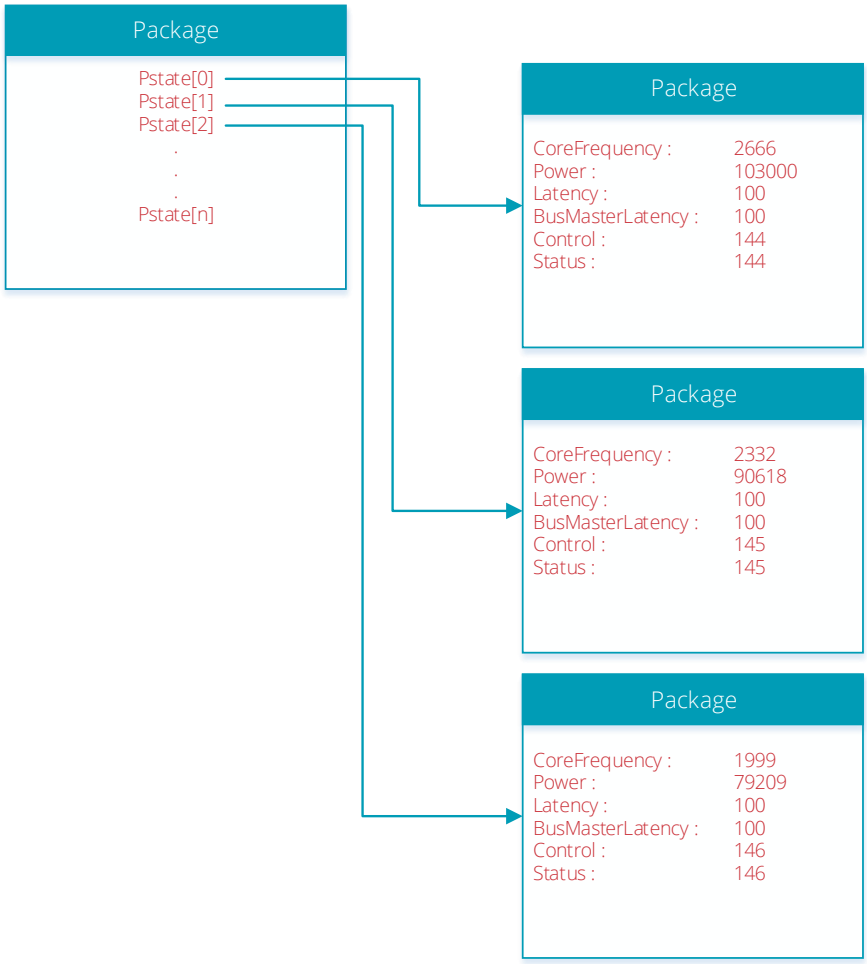


Figure A.2: The figure shows an example of a ACPI AMLpackage returned when evaluating a _PSS object. A detailed description of the different fields can be found in the ACPI specification version 5.0 [22, p. 409].

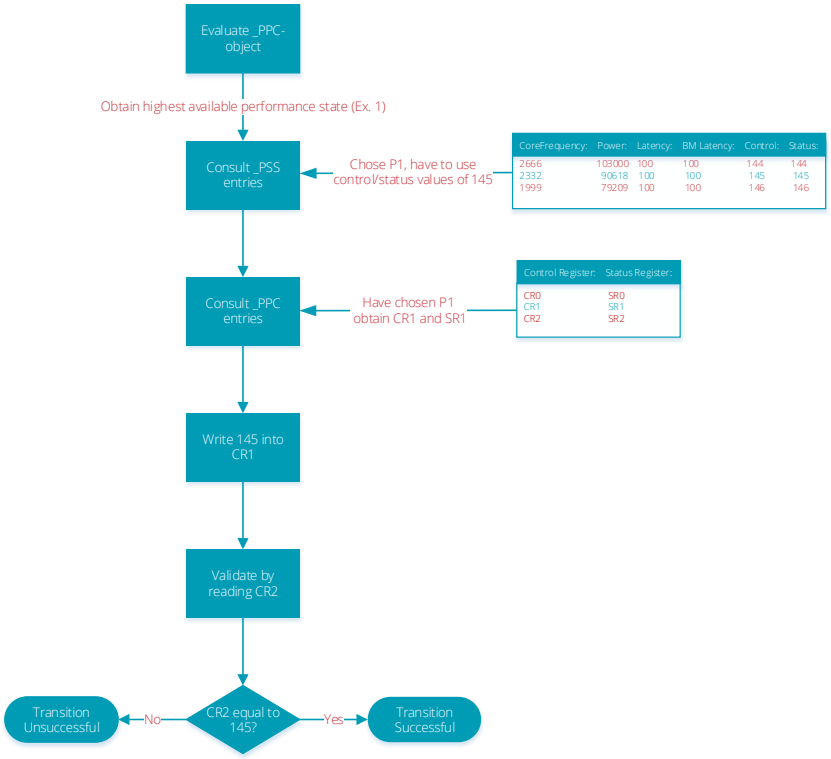


Figure A.3: The figure illustrates the process of entering a performance state.

OSPM must evaluate the _PPC object described above to obtain the currently available performance states. The control value to write can be obtained by evaluating the _PSS object. To validate that the transition was performed successfully, the returned status register must be read. If the transition was indeed successful, the value read will match the one present in the status field of the _PSS entry corresponding to the performance state.

A.2.5 The _PSD Object

The P-state dependency-object(_PSD) is used byOSPM code to obtain information regarding P-state cross logical processor dependencies. This information is crucial since different processors and platforms exhibit different dependency characteristics. For instance, different cores on a multi-core chip may or may not be dependent on each other.

A.2.6 The _CST Object

The optional _CST object provides on of two alternative methods for OSPM software to transition a processor into different processor C-states. This alternative method allows the platform designers to support more than three C-states. When evaluating the _CST object, an ACPI AML package containing an integer denoting the number of supported C-states and the same number of subpackages—each describing one of these C-states— is returned. This is illustrated in figure A.4.

The C1 state is mandatory, while all other C-states are optional. Each of the subpackages detailing a C-state contains relevant information such as the *type*(C_1, C_2, ..., C_n) determining the entry semantics, the worst-case latency when entering/exiting the state, and the average power consumption of the processor when placed in the power-state. It also contains a *register descriptor* field, which is used to determine the entry method for the C-state. For instance, which registers to use, whether these are functional fixed hardware (FFH) or system I/O registers, and exactly how to use these. A detailed description of the _CST object and resource descriptors can be found in [22, p. 395–397], and [23] respectively.

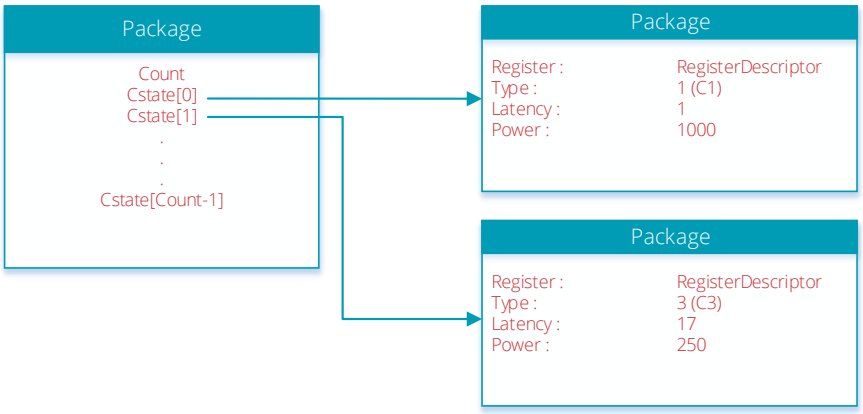


Figure A.4: The displays an example returned package from evaluation of the _CST object. Note that the C1 power state is mandatory, while all other C-states are optional.

Bibliography

[1] Greenhouse Gas Equivalencies Calculator. http://carbonfootprint360.com/p/Greenhouse-Gas-Equivalencies-Calculator.html. Accessed 2014-04-01.

[2] Linux Cross Refrence, Intel ACPI C-state driver. http://lxr.free-electrons.com/source/arch/x86/kernel/acpi/cstate.c. Accessed 2014-05-14.

[3] Overview of the TPC Benchmark C: The Order-Entry Benchmark. http://www.tpc.org/tpcc/detail.asp. Accessed 2014-05-01.

[4] H.R.5646 - To study and promote the use of energy efficient computer servers in the United States. http://beta.congress.gov/bill/109th-congress/house-bill/5646, July 2006. Accessed 2014-03-23.

[5] EU to Study Energy Use by Data Centers. http://www.pcworld.com/article/129322/article.html, February 2007. Accessed 2014-03-23.

[6] Monitor/mwait. http://blog.andy.glew.ca/2010/11/httpsemipublic.html, November 2010. Accessed 2014-05-01.

[7] Add P state driver for Intel Core Processors. https://lwn.net/Articles/536017/, February 2013. Accessed 2014-03-19.

[8] Facebooks Carbon & Energy Impact. https://www.fb-carbon.com/pdf/FB_carbon_enegergy_impact_2012.pdf, June 2013. Accessed 2014-04-01.

[9] Nevine AbouGhazaleh, Alexandre Ferreira, Cosmin Rusu, Ruibin Xu, Frank Liberato, Bruce Childers, Daniel Mosse, and Rami Melhem. Integrated CPU and L2 Cache Voltage Scaling Using Machine Learning. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 41–50, New York, NY, USA, 2007. ACM.

[10] Baris Aksanli, Jagannathan Venkatesh, Liuyi Zhang, and Tajana Rosing. Utilizing Green Energy Prediction to Schedule Mixed Batch and Service Jobs in Data Centers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[12] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, December 2007.

[13] Cullen Bash and George Forman. Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-efficient Locations in the Data Center. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 29:1–29:6, Berkeley, CA, USA, 2007. USENIX Association.

[14] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE TRANSACTIONS ON VLSI SYSTEMS*, 8(3):299–316, 2000.

[15] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The Case for Power Management in Web Servers, 2002.

[16] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 240–249, New York, NY, USA, 2008. ACM.

[17] Gavin C. Cawley. On a Fast, Compact Approximation of the Exponential Function. *Neural Comput.*, 12(9):2009–2012, September 2000.

[18] Hanen Chihi, Walid Chainbi, and Khaled Ghedira. An energy-efficient self-provisioning approach for cloud resources management. *SIGOPS Oper. Syst. Rev.*, 47(3):2–9, November 2013.

[19] Eui-Young Chung, Luca Benini, Alessandro Bogliolo, Ro Bogliolo, Yung-Hsiang Lu, and Giovanni De Micheli. Dynamic Power Management for

Nonstationary Service Requests. In *In Design Automation and Test in Europe*, pages 77–81, 2002.

[20] AMD Corporation. Cool 'n' Quiet™Technology Installation Guide for AMD Athlon™64 Processor Based Systems, Revision 0.04, Jun. 2004.

[21] AMD Corporation. Amd PowerNow!™Technology - Dynamically Manages Power And Performance, Informational White Paper, Revision A, Nov. 2000.

[22] Compaq Computer Corporation and Revision B. Advanced Configuration and Power Interface Specification, Revision 5, 2011.

[23] Intel Corporation. Intel Processor Vendor-Specific ACPI - Interface Specification, Revision 005, 2006.

[24] Intel Corporation. ACPI Component Architecture User Guide and Programmer Reference, Revision 5.17, 2013.

[25] Intel Corporation. Intel™64 and IA-32 Architectures Software Developer's Manual, Volume 2A Instruction Set Reference, A-M, 2014.

[26] Intel Corporation. Intel™64 and IA-32 Architectures Software Developer's Manual, Volume 3B System Programming Guide, 2014.

[27] Intel Corporation. Enhanced Intel®SpeedStep®Technology for the Intel®Pentium®M Processor, Mar. 2004.

[28] Gaurav Dhiman and Tajana S. Rosing. Dynamic power management using machine learning. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 747–754, New York, NY, USA, 2006. ACM.

[29] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic Voltage Frequency Scaling for Multi-tasking Systems Using Online Learning. In *Proceedings of the 2007 international symposium on Low power electronics and design*, pages 207–212. ACM, 2007.

[30] Joseph Doyle, Robert Shorten, and Donal O'Mahony. Stratus: Load Balancing the Cloud for Carbon Emissions Control. *IEEE Transactions on Cloud Computing*, 1(1):1, 2013.

[31] Frank Durr. Improving the efficiency of cloud infrastructures with elastic tandem machines. In *Cloud Computing (CLOUD), 2013 IEEE Sixth*

International Conference on, pages 91–98. IEEE, 2013.

[32] Mark E. Femal and Vincent W. Freeh. Boosting Data Center Performance Through Non-Uniform Power Allocation. In *ICAC*, pages 250–261. IEEE Computer Society, 2005.

[33] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs. *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2011.

[34] Íñigo Goiri, William Katsak, Kien Le, Thu D. Nguyen, and Ricardo Bianchini. Parasol and GreenSwitch: Managing Datacenters Powered by Renewable Energy. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 51–64, New York, NY, USA, 2013. ACM.

[35] Richard Golding, Peter Bosch, and Carl Staelin. Idleness is Not Sloth, 1995.

[36] Erlend Graff. Initial Design and Implementation of a Windows VM OS for Vortex. Bachelor thesis, Department of Computer Science, University of Tromsø, 2014.

[37] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive Caching by Refetching. In *In Advances in Neural Information Processing Systems 15*, pages 1465–1472. MIT Press, 2002.

[38] Flavius Gruian. Hard Real-time Scheduling for Low-energy Using Stochastic Data and DVS Processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 46–51, New York, NY, USA, 2001. ACM.

[39] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, page 297, 2000.

[40] Sebastian Herbert and Diana Marculescu. Variation-Aware Dynamic Voltage/Frequency Scaling. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 301–312. IEEE, 2009.

[41] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. In *Proceedings of the 4th ACM/SPEC Interna-*

tional Conference on Performance Engineering, ICPE '13, pages 187–198, New York, NY, USA, 2013. ACM.

[42] Mark Herbster and Manfred Warmuth. Tracking the Best Expert. In *Machine Learning*, pages 286–294. Morgan Kaufmann, 1995.

[43] Courtney Humphries and Paul Ruth. Towards Power Efficient Consolidation and Distribution of Virtual Machines. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 75:1–75:6, New York, NY, USA, 2010. ACM.

[44] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, pages 679–688, 2006.

[45] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.

[46] M. Ciraula E. Fang S. Johnson Bujanos R. Kumar D. Wu M. Braganza J. Dorsey, S. Searles and S. Meyers. An Integrated Quad-Core Opteron Processor, 2007.

[47] Hwisung Jung and Massoud Pedram. Supervised Learning Based Power Management for Multicore Processors. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(9):1395–1408, September 2010.

[48] Jonathan G. Koomey. Growth in Data Center Electricity Use 2005 to 2010, 2011.

[49] Jonathan G. Koomey, Christian Belady, Michael Patterson, and Anthony Santos. Assessing trends over time in performance, costs, and energy use for servers, 2009.

[50] Ramakrishna Kotla, Anirudh Devgan, Soraya Ghiasi, Tom Keller, and Freeman Rawson. Characterizing the Impact of Different Memory-Intensity Levels. In *In IEEE 7th Annual Workshop on Workload Characterization (WWC-7*, 2004.

[51] Ramakrishna Kotla, Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling Processor Voltage and Frequency in Server and Cluster Systems. In *Proceedings of the 19th IEEE International Parallel and Distributed*

Processing Symposium (IPDPS'05) - Workshop 11 - Volume 12, IPDPS '05, pages 234.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Åge Kvalnes. *The Omni-Kernel Architecture: Scheduler Control Over All Resource Consumption in Multi-Core Computing Systems*. PhD thesis.

[53] Åge Kvalnes, Dag Johansen, Robbert van Renesse, Fred B. Schneider, and Steffen Viken Valvåg. Omni-Kernel: An Operating System Architecture for Pervasive Monitoring and Scheduling. Technical Report IFI-UiT 2013-75, Department of Computer Science, University of Tromsø, 2013.

[54] Branislav Kveton and Shie Mannor. Adaptive timeout policies for fast finegrained power management. In *In Proceedings of the 19th Conference on Innovative Applications of Artificial Intelligence*, 2007.

[55] Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. Energy-efficient Virtual Machine Provision Algorithms for Cloud Systems. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 81–88, Washington, DC, USA, 2011. IEEE Computer Society.

[56] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. In *Machine Learning*, pages 285–318, 1988.

[57] Nick Littlestone and Manfred K. Warmuth. The Weighted Majority Algorithm. *Inf. Comput.*, 108(2):212–261, February 1994.

[58] Vimal Mathew, Ramesh K Sitaraman, and Prashant Shenoy. Energy-aware Load Balancing in Content Delivery Networks. 2011.

[59] David Meisner, Brian T. Gold, and Thomas F. Wenisch. The PowerNap Server Architecture. *ACM Trans. Comput. Syst.*, 29(1):3:1–3:24, February 2011.

[60] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[61] Michael Moeng and Rami Melhem. Applying Statistical Machine Learning to Multicore Voltage & Frequency Scaling. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 277–286, New York, NY, USA, 2010. ACM.

[62] Fereydoun Farrahi Moghaddam, Mohamed Cheriet, and Kim Khoa

Nguyen. Low carbon virtual private clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 259–266. IEEE, 2011.

[63] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, August 1997.

[64] Justin Moore, Jeff Chase, Parthasarathy Ranganathan, and Ratnesh Sharma. Making Scheduling "Cool": Temperature-aware Workload Placement in Data Centers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[65] Ripal Nathuji and Karsten Schwan. Reducing system level power consumption for mobile and embedded platforms. In *In Proceedings of the International Conference on Architecture of Computing Systems (ARCS*, 2005.

[66] Ripal Nathuji and Karsten Schwan. Virtualpower: Coordinated Power Management in Virtualized Enterprise Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 265–278, New York, NY, USA, 2007. ACM.

[67] Audun Nordal, Åge Kvalnes, and Dag Johansen. Balava: Federating Private and Public Clouds. In *2011 IEEE World Congress on Services*, pages 569–577, 2011.

[68] Audun Nordal, Åge Kvalnes, and Dag Johansen. Paravirtualizing TCP. In *6th international workshop on Virtualization Technologies in Distributed Computing*, pages 3–10, 2012.

[69] Audun Nordal, Åge Kvalnes, Robert Pettersen, and Dag Johansen. Streaming as a Hypervisor Service. In *7th international workshop on Virtualization Technologies in Distributed Computing*, 2013.

[70] G. A. Paleologo, L. Benini, A. Bogliolo, and G. De Micheli. Policy Optimization for Dynamic Power Management. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 182–187, New York, NY, USA, 1998. ACM.

[71] C. D. Patel, C. E. Bash, R. Sharma, and M. Beitelmal. Smart cooling of data centers. In *Proceedings of IPACK*, 2003.

[72] Padmanabhan Pillai and Kang G. Shin. Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems. *SIGOPS Oper. Syst. Rev.*, 35(5):89–102, October 2001.

[73] Cai Qiong, José González, Grigorios Magklis, Pedro Chaparro, and Antonio González. Thread Shuffling: Combining DVFS and Thread Migration to Reduce Energy Consumptions for Multi-core Systems. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 379–384. IEEE, 2011.

[74] Karthick Rajamani, Heather Hanson, Juan Rubio, Soraya Ghiasi, and Freeman Rawson. Application-Aware Power Management. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 39–48. IEEE, 2006.

[75] Dinesh Ramanathan and Rajesh Gupta. System Level Online Power Management Algorithms, 2000.

[76] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 66–77, Washington, DC, USA, 2006. IEEE Computer Society.

[77] Zhiyuan Ren, Bruce H. Krogh, and Radu Marculescu. Hierarchical Adaptive Dynamic Power Management. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10136–, Washington, DC, USA, 2004. IEEE Computer Society.

[78] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1*. Microsoft Press, Redmond, WA, USA, 6th edition, 2012.

[79] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, Redmond, WA, USA, 6th edition, 2012.

[80] Nicol N. Schraudolph. A Fast, Compact Approximation of the Exponential Function. *Neural Computation*, 11:11–4, 1998.

[81] Mohsen Sharifi, Hadi Salimi, and Mahsa Najafzadeh. Power-efficient distributed scheduling of virtual machines using workload-aware consolidation techniques. *The Journal of Supercomputing*, 61(1):46–66, 2012.

[82] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, Kevin Skadron, and Zhijian Lu. Power-aware QoS Management in Web Servers. In *Proceedings*

of the 24th IEEE International Real-Time Systems Symposium, RTSS '03, pages 63–, Washington, DC, USA, 2003. IEEE Computer Society.

[83] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Chip Multi Processing aware Linux Kernel Scheduler. In *Linux Symposium*, page 329, 2006.

[84] Tajana Simunic, Luca Benini, Peter Glynn, and Giovanni De Micheli. Event-Driven Power Management. *IEEE TRANS. COMPUTER-AIDED DESIGN*, 20:840–857, 2001.

[85] Mani B. Srivastava, Anantha P. Chandrakasan, and R. W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Trans. Very Large Scale Integr. Syst.*, 4:42–55, March 1996.

[86] Carl W. Steinbach. A Reinforcement-Learning Approach to Power Management, 2002.

[87] The Climate Group. SMART 2020: Enabling the low carbon economy in the information age. Technical report, 2008.

[88] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the Energy Efficiency of a Database Server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 231–242, New York, NY, USA, 2010. ACM.

[89] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[90] David Wang. Meeting green computing challenges. In *Electronics Packaging Technology Conference, 2008. EPTC 2008. 10th*, pages 121–126. IEEE, 2008.

[91] Andreas Weissel and Frank Bellosa. Process Cruise Control: Event-driven Clock Scaling for Dynamic Power Management. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 238–246, New York, NY, USA, 2002. ACM.

[92] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

[93] Rong Ye and Qiang Xu. Learning-Based Power Management for Multi-Core Processors via Idle Period Manipulation.

[94] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 124–134, New York, NY, USA, 1992. ACM.

[95] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 149–163, New York, NY, USA, 2003. ACM.