UNIVERSITY OF CALIFORNIA
RIVERSIDE

Thermal and Power Management for Network and Data Center Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Chih-Hsun Chou

September 2017

Dissertation Committee:

Dr. Laxmi N. Bhuyan, Chairperson
Dr. Shaolei Ren
Dr. Daniel Wong

The Dissertation of Chih-Hsun Chou is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

It is my great pleasure to thank those who made this dissertation possible. I would never have been able to finish it without the guidance of my committee members, help from friends, and support from my family.

I would like to express my deepest gratitude to my advisor, Dr. Laxmi Bhuyan, for his excellent guidance, caring, patience, and providing me with a pleasant atmosphere for doing research. I would also like to thank Dr. Daniel Wong and Dr. Shaolei Ren for guiding my dissertation, giving precious advice and participating in my final defense committee.

I would like to thank my dear wife and the entire family. They were always supporting me and encouraging me with their best wishes.

I would like to thank all members from my lab, who were always willing to help, discuss ideas, and give helpful suggestions. It would have been a lonely lab without them. Many thanks to all my collaborators and friends for helping me complete my Ph.D study. Without them, it would have been difficult to write this dissertation.

This dissertation includes content published in the following journals and proceedings:

ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2013

Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems (ANCS), 2014

33rd IEEE International Conference on Computer Design (ICCD), 2015

Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), 2016

IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017

ABSTRACT OF THE DISSERTATION

Thermal and Power Management for Network and Data Center Applications

by

Chih-Hsun Chou

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, September 2017
Dr. Laxmi N. Bhuyan, Chairperson

The last decade has brought an explosive growth of delay-sensitive interactive services that have become an integral part of our lives and constituted an increasingly high portion of network and data center workloads. To attract users and generate revenue, interactive services require high-quality and timely responses. Further, these applications are typically deployed over a large set of high performance servers consuming a significant amount of power/energy. The high power/energy consumption not only directly impacts the operation cost, it also results in high operating temperature, which incurs exponentially increased cooling cost and performance degradation.

In this thesis, we focus on developing power management techniques for network and data center applications. We propose various techniques to reduce the power/energy consumption while satisfying different performance constraints. First, we propose a per-core power management technique based on CPU sleep states for various network applications. The packets are queued in a buffer so that the core can sleep longer. In order to limit the operating temperature, we control the duration that a core can be active and inactive.

Moreover, we develop a heterogeneous load distribution algorithm to achieve better trade-off between thermal behavior and power saving. Second, due to random nature of packet arrivals and packet processing times, satisfying the packet delay constraint is challenging. By developing statistical performance models, we develop a runtime technique to determine when, how long and which cores should be inactive to reduce the power consumption.

Unlike network applications, performance of interactive applications is defined by strict tail latency constraints. With the fast-varying traffic patterns, to satisfy the tail latency constraint, we first propose a dynamic sleep scheme which adjusts the wakeup time of the CPU cores based on request arrivals. Followed by a detailed performance analysis, we conclude that the state transition overhead is another source of energy inefficiency. We propose an all-encompassing power management technique, called μDPM, which coordinates sleep, speed scaling and request dispatching to reduce active, idle and state transition energy consumption all together. Finally, we consider a web search application and observe that the result quality and tail latency together determine the system-wide performance and energy consumption. We explore the application characteristics and propose a quality and latency aware power management technique by judiciously discarding long query executions with ISN-aggregator coordination. Through extensive experiments, we conclude that our schemes significantly reduce the energy consumption while satisfying both the quality and tail latency constrains.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the explosive growth in the amount of data over the Internet, the network infrastructure and content providers require orders-of-magnitude increase in the capability to meet the throughput and QoS requirement of users. The applications range from network packet processing to the servers in data centers that serve the users requests. The general purpose multicore processors provide the flexibility and high performance to deploy various packet processing algorithms, such as IP forwarding [41], packet classification [89] and deep packet inspection [101], and are commonly deployed in the network infrastructure. Similarly, they form the basic server architecture for data center applications, such as memcached [7, 47] and web search [1,47]. Usually, the network and data center infrastructures are designed with over provisioning to meet the QoS demands, giving rise to low energy proportionality. Thus, increasing the performance and decreasing the power consumption are first-order design goals of modern network and datacenter servers.

We consider network packet processing and data center applications under the same umbrella because the requests arrive at a server and are served non-deterministically. The service time and arrival pattern vary from request to request and application to application. Along with the high throughput, power consumption plays an increasingly important role in network and datacenter applications. Due to the varying traffic load and non-deterministic request arrival pattern, current multicore servers for these applications lack *energy proportionality* – the power consumption does not scale down proportionally with the traffic load. Because the service capability is provisioned with peak traffic demand, significant amount of energy is wasted during the medium and low traffic periods.

Both network and datacenter applications exhibit ON/OFF patterns [23], as shown in figure 1.1. The x-axis shows the random packet arrivals to a server and y-axis shows the work in queue. The power consumption of the server is shown in dotted red line. The CPU core consumes active power during the ON period (also known as *busy period*) when requests are being processed. During the OFF period (or *idle period*), where no request arrives, the CPU consumes idle power, which is less than the active power, but still quite high. The OFF period between batches of requests provides opportunities to perform

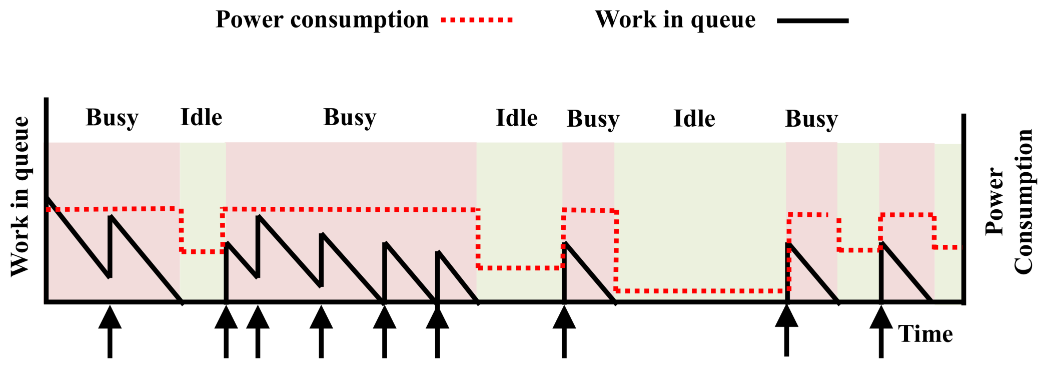


Figure 1.1: The packet/request processing and power consumption pattern.

power management. It is important to note that the length of ON/OFF period is not deterministic, and depends on the request arrival and service rates. At the hardware level, two main techniques are used for reducing the energy consumption: (1) Dynamic Voltage and Frequency Scaling and (2) CPU Sleep. The first technique, Dynamic Voltage and Frequency Scaling (DVFS), reduces or increases processor voltage/frequency just enough to meet the performance requirement. DVFS can be either chip-wide, where the entire chip is scaled as one unit, or per-core, where individual cores on the chip can be scaled at different rates. Per-core DVFS adjusts the frequency of each core individually at runtime with on-chip voltage regulators. Although it is more complex to implement, per-core DVFS achieves greater power savings with its fine-grain control over individual cores. DVFS has been applied for network packet processing and data center applications [23, 76, 78, 80, 92]. However, two main physical limitations greatly affect the effectiveness of DVFS. First, the operating voltage approaches the transistor threshold voltage as technology advances, hence decreasing the room for voltage down-scaling [93]. Second, DVFS focuses on reducing the dynamic power consumption only, but static power consumption has become more dominant in today's multicore processors with more advanced manufacturing process [66, 72]. These limitations can not be overcome without some breakthrough in the transistor design.

The second technique, CPU sleep, minimizes leakage current when a core is idle by powering down different CPU components. Modern CPUs provide succession of per-core idle states (called sleep states or C-states), where different CPU core components are turned off. The transition time of a sleep state depends on the degree of sleep. The deeper the sleep state, the more time it takes to enter/exit. Because of the state transition overhead, it only makes sense to enter a C-state if inactive time is beyond a certain threshold. Some work proposed to put the entire processor/device into idle states during periods of inactivity [76, 91]. Also, a few papers considered switching off a few CPU cores depending on the throughput requirement [48, 80, 92]. While it is promising on reducing energy consumption, some application execution characteristics limit the use of CPU sleep state for energy reduction. First, for parallel network/datacenter applications, the idle periods of individual cores, each serving independent jobs, hardly ever align. As a result, the effectiveness of processor idle low-power modes is reduced because of the needs of frequent waking up. The processor-wide idleness rapidly vanishes as the traffic load or number of cores grows. Also, waking up the cores from processor level idle states takes a long time for which the OS does not drive them to switch off modes during normal operation. Second, because of the state transition energy overhead, the natural idle periods are not long enough for the OS to put cores into the processor level deep sleep states.

Several researches have been done on reducing the energy consumption by turning off the servers. In [100], “buffer and burst” is proposed to turn off the network router for a fixed amount of time and its impact on the energy consumption and packet latency is investigated. However, this approach focuses on a single device and does not address the problem in multiple cores/servers scenario. Also, it still suffers from the limitation of long state transition latency. In [48], a similar situation of multiple servers with active and inactive states is analyzed. This work focuses on developing various policies to determine when and which sleep states the server should enter. However, they focus on the centralized queue scheme, where off state is a natural extension of the sleep states. The multicore packet processing is usually modeled as distributed queue scheme, where off and sleep states differ significantly. Second, they did not incorporate various C-states in the CPU that suffers from long transition overhead or practically verify the analysis.

Besides power and energy, temperature and thermal constraints also pose a significant challenge to future system design. As the size of the multicore chip decreases with the advancement of the manufacture process, the power density increases significantly. This poses a major challenge toward thermal management in order to maintain the chip temperature within the operating limit. It is known that high operating temperature adversely affects not only the system performance and leakage power, but also the throughput [43], circuit reliability and chip lifetime [111]. In addition, the cooling and packaging cost for heat dissipation increases exponentially with the peak temperature [110]. In the "run-to-finish" scientific applications, the CPU always consumes active power and generates heat while running, and its temperature simply rises to saturation point until it stabilizes. The ON/OFF pattern of network applications exhibit a sawtooth temperature behavior, where the CPU core temperature rises during the ON period, and falls during the idle period. Although reducing the busy/idle period power consumption through DVFS and CPU sleep will help slow down the temperature rise and speed up the CPU cooling, the *length* of busy/idle period will also need to be considered while designing thermal aware scheduling algorithm. Moreover, in multicore CPU, since the cores are densely packed, the temperature variation of a core will have great impact on its neighboring core due to the horizontal heat flows. These complex thermal interactions make the problem challenging in multicore thermal management.

In chapter 3, we present a novel power aware thermal management algorithm to achieve power saving in multicore processors by employing a *vacation* scheme. In vacation scheme, we focus on controlling the duration of both active and sleep periods. We force the core to sleep for a fixed amount of time in different sleep states. We implement the scheme through the idle states (C-state) provided by the OS in the CPU and show their effectiveness both through analysis and experimental data. Then, we apply thermal constraint to our scheme. To predict the performance, we develop the multicore thermal RC models considering the temperature coupling among cores. Moreover, we derive the power consumption model considering the CPU C-states, and latency model through vacation queuing theory [113]. By combining these models, we can accurately depict the performance with our multicore vacation scheme. Due to the non-linear performance behaviour with different traffic loads, we propose a heterogeneous load distribution, which creates more opportunities for power saving. Besides maintaining processor temperature below the temperature constraint, our technique achieves higher sustainable load and better power saving with minimum latency increase compared to existing thermal management techniques. In chapter 4, we propose a multicore idle period power management scheme by dynamically adjusting the per-core sleep time with different traffic. During the sleep period, the core can enter deep idle state without being waken up prematurely. We call this "smart sleep", when the core sleeps and the arriving requests are queued. Moreover, we explore the possibility of completely turning off some CPU cores by redistributing the traffic onto a subset of cores. By doing so, minimizing the power consumption becomes a two-dimensional optimization problem with both time (sleep) and space (off) variations.

We derive the expected idle and busy periods by considering the length of the core sleep, the number of inactive cores, request service time and the traffic load. By combining these models, we can find the behavior of power consumption under our proposed scheme. Also, we derive the response time model using the M/G/1 queuing model with vacation [113]. Our proposed technique overcomes the limitations that currently exist in the CPU sleep power management scheme, and provide better power-performance trade-offs compared with the existing techniques. The experimental results show that our proposed technique can achieve more than 35% power saving and 60% increase in energy proportionality under real world traffic trace compared with the existing power management techniques.

While throughput remains critical performance requirement for network applications, latency-critical applications are becoming increasingly common in current data centers and pose new challenges for system designers. They exhibit similar ON/OFF characteristics as network applications. Moreover, tail latency, not average latency, determines the quality of service for these applications. Servers running latency-critical workloads are kept lightly loaded to meet strict latency targets, with typical utilization between 10% and 50% [21,91]. This low utilization wastes billions of dollars annually because servers are not energy-proportional and have poor energy efficiency at low server utilization [21,69,86,117].

Although servers are usually kept lightly loaded, peak load is considered when setting the proper target tail latency. Thus, the tail latency of servers running under low load will be far lower than the target tail latency. Figure 1.2 shows the tail latency (95th percentile latency) of a server running the Memcached application [7] under different loads. We can see that there exists a "latency slack" between the tail latency of the server (solid

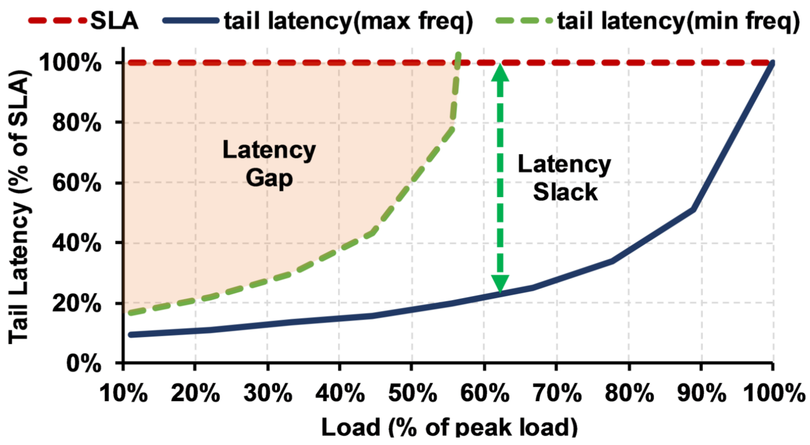


Figure 1.2: Significant Latency Gap Exists

black) and the target tail latency as defined in the service level agreement (SLA)(dotted red line). This latency slack provides opportunities for achieving energy proportionality.

Several DVFS-based techniques [69, 85, 86, 114] have been proposed to exploit this latency slack to slow down processing so that the requests finish just before the target tail latency. However, their benefits are limited due to the "floor" of frequency scaling. Figure 1.2 shows the tail latency at minimum frequency(dotted green) and at maximum frequency(solid black). Clearly, DVFS can close the latency slack at higher load above ~50%. However, there still exists a significant "latency gap" at 10%~50% even under the minimum frequency. This low utilization range is where most data center servers spend most of their time [21, 91]. Clearly, there exists significant opportunity for power savings by targeting this latency gap. Other prior works exploit CPU sleep states by batching and delaying the processing of requests to create idleness [44, 91, 93]. However, these techniques do not adhere to strict SLA levels and are not suitable for latency-critical applications.

In data centers, traffic pattern often exhibits high short-term variability. Requests arrive at unpredictable times and are often bursty. These short-term spikes dominate tail latency [64, 69] and must be accounted for. To address these issues, in chapter 5, we focus on developing the power management techniques with the consideration of this traffic variability. To exploit the opportunities of power saving with latency slacks, First, we propose a dynamic sleep scheme, a fine-grain power management scheme that exploits latency slack by dynamically postponing request processing. By doing so, the idle periods between two active periods will be prolonged, and the CPU core can efficiently enter deep sleep states to save power. Unlike previous works that require some hardware changes, such as per-core DVFS with nanosecond level transition time [69] and fast sleep state transition techniques [91,93], our proposed technique uses the commonly supported per-core sleep state. Then, we perform a detailed analysis for both sleep and DVFS based power management schemes and find out that the existing dynamic power management techniques are largely ineffective because of the high state transition overhead, short request service time and strict tail latency constraint. Our analysis also shows that sleep or DVFS alone can not fully exploit the latency slack for power saving. We take the lessons learned here to design μDPM, an all encompassing power management framework that coordinates request delaying, per-core sleep states, voltage frequency scaling and load distribution. μDPM extends our dynamic sleep scheme by taking into the account of request processing speed. Also, we develop an novel request redirection algorithm which aims to minimize the both sleep and DVFS state transitions, which are the major sources of energy inefficiency while applying sleep and DVFS power management.

In the data centers, hundreds of thousands of servers work together to achieve strict performance requirements. Instead of per server performance, the system-wide performance is becoming more and more important for online data-intensive applications (OLDI), such as Web Search, social networking and e-commerce. Needless to say, these applications need to meet strict performance constraints, which directly affect users satisfaction as well as revenues [108]. These applications need to process tens of thousands of requests every second over billions of documents, but users expect the response time to be under a few hundred milliseconds. Although high-quality to users queries are crucial in search results, the service provider may often have to compromise the quality in order to meet the stringent latency requirement. Additionally, the system-wide power consumption is also a critical performance metric. With hundreds of thousands of machines, significant power consumption is already leading to a soaring operating cost for the search service provider. As a result, reducing power consumption while meeting the quality and latency constraints is a major challenge in designing OLDI system.

Query processing in OLDI applications often share a common characteristic: They employ a multi-level tree-like software architecture where each query first goes to the aggregator and then goes to all or many index serving nodes (ISN). Since the aggregator needs to wait for all responses from the ISNs, the response time depends on the slowest ISN. Due to significant response time variation among the ISNs, a long latency at any ISN manifests as a slow query response at the aggregator level [18, 37, 71]. At each ISN, returning top-K results requires exhaustively computing the relevance score for each indexed document. This is time consuming and often unnecessary since most of the evaluated documents will not make to the ISN-level top-K results. Moreover, most of the results at ISNs are not used because only the top-K results from the aggregator will be sent back to the user. Clearly, these unnecessary computations, albeit useful for producing high-quality search results, lead to long query response time and power wastage.

Several techniques have been proposed to reduce request processing time at the ISN level. For example, dynamic pruning (or early termination) reduces the number of documents to be evaluated by skipping the documents that can not make to the top-K results [39,40]. Although the pruning methods reduce some unnecessary computations and improve the average query processing time at the ISN, the query processing time variation among ISNs is still high [65]. As a result, the improvement of the query response time at the aggregator level is limited. Another method is proposed to employ an aggregator timeout, which determines how long the aggregator waits for its ISNs before sending the results back to users [120]. The key idea is to trade off the result quality and the response time: the shorter the aggregator waits, the less ISN response it collects, which improves responsiveness but potentially degrades the quality due to missing of results from certain ISNs. The aggregator policies improve the response time subject to a quality constraint, but does not address the power consumption issue since the ISNs still perform the query processing for all queries with unnecessary power wastage. Further, [120] defines the quality as the number of responses received from the ISNs, but this does not capture the characteristic of a distributed search engine that different ISN responses may contribute to the final result differently due to ISN heterogeneity.

To satisfy quality and latency constraint in the search system, in chapter 6,we develop a cooperative aggregator-ISN policy that cuts out the slow ISN responses for fast response time and reduces energy consumption. First, at the aggregator, a “time budget” is assigned to the ISNs for request processing based on the expected quality level. Based on the budget, each ISN serves the request only if the predicted processing time is within the time budget, and disregards the request for energy saving otherwise. Furthermore, the aggregator may choose to re-send the query for full execution in case the response quality is not satisfactory. By doing so, we essentially reduce the ISN processing workload and aggregator level latency, while still meeting the quality constraint. Since the time budget is known to the ISNs, we propose to proactively put selected ISNs to sleep state to save power when the time budget cannot be met.

The rest of this thesis is organized as follows. Chapter 2 presents related work and motivation for our work. In chapter 3, we introduce thermal-aware power management scheme. In chapter 4, we propose a statistical model based latency constrained power management scheme. Chapter 5 focuses on latency-constrained power management technique for data center applications. We propose a system-wide quality and latency-aware scheduling scheme for distributed OLDI applications in chapter 6. Finally, chapter 7 concludes this thesis.

Chapter 2

Related Work

2.1 Power Management Techniques

Much work has been done on reducing power/energy consumption of computer systems. Based on the techniques, we can classify these works into three categories: (1) Dynamic Voltage Frequency Scaling (DVFS), (2) CPU sleep state and (3) Dynamic resource provisioning.

The first group utilize the DVFS to slow down the processor processing speed to reduce power/energy consumption. [50] minimizes energy usage by choosing the best combination of voltages for each task based on the task dependency. [107] applies genetic list scheduling algorithms (GLSA) to schedule and map tasks, considering the processor's power profile during a refined voltage section. Some works focus on differentiating program execution phases based on different CPU/memory-bound ratios and apply DVFS accordingly, because reducing processor frequency when the program is in the memory-bound phase will not affect performance. Specifically, [60] analyzes various global power management policies using per-core DVFS to maximize performance for a given power budget. In [61], they propose a runtime phase predictor that works cooperatively with DVFS. [35] regulates concurrency and changes processors/threads configuration as the program executes by hardware event driven profiling. In [78], Kuang et al. use DVFS for scheduling of networking applications. This scheme allocates frequencies to the pipeline stages statically and is not aimed at exploiting different traffic variations. However, two main physical limitations greatly affect the effectiveness of DVFS. First, the operating voltage approaches the transistor threshold voltage as technology advances, hence decreasing the room for voltage and frequency down-scaling [93]. Second, DVFS focuses on reducing the dynamic power consumption only, but static power consumption has become more dominant in today's multicore processors with more advanced manufacturing process [66, 72]. These limitations drive the research communities to develop the power management technique without relying on DVFS.

In the second category, CPU sleep reduces static power when a core is idle by powering down different CPU components. Some works proposed to forced the core into deep sleep state when idle. In [76, 91], the authors proposed to put the network devices into power saving mode during the idle period, and wake up the processor when a packet arrives. In [91, 93], the CPU enters the deep sleep state when no pending task exists. In [100], the authors proposed "buffer and burst" to throttle the execution. It buffers the incoming tasks for a period of time, during which the devices will stay idle. With this approach, the incoming traffic is reshaped into small bursts of arrival; devices wake up to transmit a burst of packets, and then sleep until the next burst arrives. The intent is to provide sufficient time for devices to enter the power saving mode with lower power consumption. However, these approaches use server/device level sleep and are highly depends on the task arriving pattern. it only achieves good energy efficiency when the task interarrival time is long enough to compensate the long (10-100 seconds) sleep state transition overhead. On the other hand, some works proposed to select the best sleep state for each idle period. In [85], the sleep state the core will enter is determined through design space exploration with the simulator based on the previous task arrival pattern. [98] uses a stochastic model to predict the sleep state a core will enter based on the workload characteristic. However, these approaches does not consider the network and data center applications, where the length of the idle period mostly falls within 100 μseconds, and can only enter shallow sleep state.

In the third category, observing that the resource demand may vary dynamically, either because of the traffic variation or workload characteristic, several works proposed to dynamically "turn-off" the hardware resources, such as the number of server and CPU cores, to reduce static power/energy consumption. In [48], the number of server staying responsive is dynamically determined based on analytical model and traffic demand. To achieve better energy efficiency in multicore processors, it is necessary to combine the dynamic resource provisioning with DVFS and sleep states. achieve fine grained power management. In [80], the authors developed an analytical model to determine how many cores can be turned off, and assign different frequency to the active cores with per-core DVFS. CARB [122] is proposed to use a C-State power management arbiter which dynamically turn off the core based on a feedback-based controller. However, the active cores still utilize the default

Linux C-state governor which performs poorly under network and datacenter applications. Much is still not known about whether deferring execution and idle states are useful in the multicore processor. Our work takes a unified approach, modeling a general multicore system with many low-power sleep states having different characteristics. It seeks to jointly manage the configuration of the number of active cores and the choice of both the length of sleep period and which sleep state to enter, and more importantly, how these decisions should be made online with the existence of the QoS requirement.

2.2 Thermal aware power management technique

Much work has been done in thermal-related research area. We classify related work in this section into two groups based on their approaches in thermal management design: (1) task migration, and (2) power reduction. Below, we introduce past work in each group.

2.2.1 Task Migration

In a multi-core single-chip processor, the temperature of a core is not only determined by the power consumption of the software program running on it but also on its heat dissipation ability and the temperature of its neighbors. The basic concept of temperature-aware task migration is to dynamically move "hot" processes to cores with superior heat dissipation and/or cores surrounded by cooler neighbors while moving "cool" processes to cores with inferior heat dissipation and/or cores surrounded by hotter neighbors.

[49] proposes heat-and-run SMT thread assignment to increase processor-resource

utilization by co-scheduling threads that use complementary resources and heat-and-run CMP thread migration to migrate threads away from overheated cores to alternate cores. [30] investigates the trade-offs between temporal and spatial hot spot mitigation schemes and thermal time constants, workload variations and microprocessor power distribution. By leveraging spatial and temporal heat slacks through task migration, their thermal-aware scheduling schemes enable lowering of on-chip unit temperatures. In [96], they propose and study a thread migration method that maximizes performance under a temperature constraint, while minimizing the number of migrations and ensuring fairness between threads. While the above mentioned techniques are reactive approaches, it requires temperature monitor and may lack responsiveness based on the monitor period. Several works proposed predictive power migration to reduce spatial and temporal temperature difference by redistributing the heat generating locations. [119] propose a thermally-aware task migration policy called Predictive Dynamic Thermal Management (PDTM). When a task running on a processor is projected to exceed the temperature threshold, it will be moved to a processor that is predicted to be the coolest in the future. [17] proposed a light weight thermal predictor through band-limited property of the temperature frequency spectrum, and migrate the task among different CPU socket to create a thermal balance and reduce peak temperature. [79] built a novel predictive model based for the periodic tasks, and apply task migration to achieve thermal balancing under given temperature constraints.

2.2.2 Power Reduction

While task migration focus on running hot processes on cooler core, the power reduction techniques try to decrease the heat generation and lower the peak temperature.

[27] is the first paper to investigate different thermal management techniques through DFS and DVFS. [110] applies control-theoretic techniques to control DVFS to avoid thermal emergencies while minimizing performance loss. [115] proposed a chip-level power control algorithm that is based on optimal control theory. Their algorithm can precisely control the power of a CMP chip through DVFS to the desired set point while maintaining per-core temperature below a specified threshold. In [116], they develop a thermal model called Matrix Model to derive core temperature and present a novel slack allocation algorithm using DVFS to minimize peak temperature. [52] proposes to use both DVFS and task-to-core allocation schemes to optimally increase the performance of multicore processors under thermal constraints. [82] proposes to optimize throughput by applying both per-core power gating and DVFS to power and thermal-constrained multicore processors. While the above techniques use DVFS to reduce the power and heat generation during the processing, it has same limitations as we discussed in the previous section. Several works proposed to throttle the CPU execution to reduce peak temperature. [19] proposed to idle cycle injection to stop-and-resume the CPU execution, during the stop period, the CPU cores will be power gated. [109] proposed to use formal feedback control and idle cycle injection to decrease thermal emergencies. However, these techniques does not consider the long state transition latency entering/exiting power gating states.

While all these works focus on reduce the heat generation, no work exploit the possibility of speed up the temperature falling in the period tasks. [79] proposed a stop-and-go technique which force CPU core to be idle and cool down its temperature. However, it assumes that the core will consumes no power during idle, which might not be the case when the idle period is too short. Also, it does not consider the impact on packet latency. Our work focus on investigating the complex performance trade-offs among temperature, power consumption and latency in network applications with the force idle technique.

2.3 Tail latency-constrained power management technique

Several techniques [69, 85, 86, 114] have been proposed to exploit latency slack by applying dynamic voltage and frequency scaling (VFS). Pegasus [86] takes a cluster-level feedback-based approach by monitoring tail latency periodically and adjusting frequency every few seconds to keep tail latency within a given bound. Pegasus is only able to exploit latency slack during low utilization. Building on this, TimeTrader [114] is able to exploit the latency slack at the request-level, rather than at the cluster-level. Observing that there exists not only server but also network latency slack, TimeTrader can save energy even during high utilization exploiting network slack and and identifying sub-critical requests to slow down using VFS. Rubik [69] proposed to save power by dynamically adjusting the frequency at request arrival and departure events. By observing the number of queued requests, Rubik pick a minimum frequency that will satisfy the tail latency constraint of every request in the system. All these prior techniques focus on VFS policies, and do not rely on sleep states. However, with technology scaling, the dynamic range of frequency scaling has decrease, along with static power consumption growing dominant [72].Recognizing this, SleepScale [85] combines a history-based predictive VFS scheme with optimal C-state selection (based on runtime simulation). Unlike the aforementioned techniques, SleepScale cannot close the latency slack as VFS settings are determined by coarse-grain prediction of future workloads. As data center traffic tend to have short-term variability, the frequent mispredictions can violate the tail latency target.

Other works utilize sleep states only to achieve power savings [91, 93]. PowerNap [91] proposed a server level fast sleep transition technique and put the server into a deep sleep state as soon as the server becomes idle, taking advantage of full-system millisecond-level idleness. Dreamweaver [93] extends PowerNap by putting entire multicore processors into deep sleep state by aligning the idle periods of cores through job preemption. While achieving good power saving, the latency overhead introduced by aligning idle periods and job preemption, results in as much as 3x tail latency increase for 40% power savings. Unfortunately, as server core count increases, full-system idleness disappears, limiting the effectiveness of Dreamweaver. While proposing sleep state techniques in chapter 4.3 and 3, they are unresponsive to short-term variability due to the fact that statistical queuing models is meaningful only when the traffic is stable for a long period of time. To address these issue, we propose a dynamic sleep scheme to determine the sleep period at the request level. Furthermore, with detail performance analysis on these state-of-the-art techniques, we are the first to illustrate that state transition overhead is the major source of energy inefficiency, and proposed a technique to combine VFS, sleep state and request redirection to minimize energy consumption with the tail latency constraint.

2.4 Quality and latency aware power management technique

In the previous section, several techniques are proposed to reduce the energy consumption by exploiting the latency slack. They does not focus on reducing the tail latency and considering the result quality in a system-wide manner. Several works proposed to use adaptive parallelism [54, 65] and computation sprinting [57, 124] to reduce query latency. In [65], the query processing will be perform in parallel when it is predicted to have long processing time. In [54], the degree of parallelism of query processing is dynamically increasing. The longer a request executes, the more parallelism it adds, hence reduce the latency of the long running requests. In [57], authors proposed to boost the operating frequency when a query identified as long query, which increase the computation power and reduce the query latency. Although these techniques improve query latency, they require additional resources. The trade-off with other system level performances, such as power consumption, is yet unclear.

Other work proposed to trade the query quality for latency reduction. Servers in the web search system often employ dynamic pruning to early terminate the postings list traversing of a query and to avoid scoring the postings for the documents that unlikely make the top-k retrieved set [39,40]. Early termination is an example that reduce latency by trading the completeness of query response [40]. [55] quantifies the relationship of quality and latency at ISNs, and trades quality for latency under heavy loads. For the aggregator level latency reduction, Yun, et.al [120] observed the latency heterogeneity among leaf servers and proposed an aggregator policy where results are sent back to users without waiting all servers' responses. By categorizing queries' behaviour among servers, the aggregator will adjust its waiting period to achieve a good latency-quality tradeoff. While being effective on reducing tail latency, it did not address the system-wide energy efficiency which is the major performance matrix in modern search system design.

While the above mentioned works focusing on quality-latency trade-off, no work has been done on simultaneously considering quality, latency and energy efficiency. In [42] exploited the quality-latency relationship and proposed a energy efficient scheduler to trade the quality for energy consumption. However, this work neither addresses the latency constraint commonly exists in the datacenter, nor provides system-wide performance analysis. To address this issue, we propose a quality and latency aware power management technique which minimizes the system-wide energy consumption while satisfying both quality and latency constraint.

Chapter 3

Power and thermal Management on Network Applications

3.1 Introduction

In this chapter, we address the thermal management issue for network packet processing. We aim at optimizing power consumption and throughput under given thermal constraint by appropriately applying per-core smart sleep. The intent is to reshape packet processing pattern and force longer CPU core idle periods to create opportunities for a CPU core to enter deeper C-states and cool down the core faster. Also, we limit the length of busy periods which prevents the core from overheating. We name our technique as a vacation scheme denoting that the server (core) has taken a vacation for a period, and the arriving packets must wait in a buffer to be processed later. This enables the core to enter into a deep sleep mode. By adopting our proposed vacation schemes, we can reduce the heat generation, power consumption, and allow the temperature to drop during idle (vacation) periods. This will avoid the peak temperature exceeding the thermal constraint. We design two vacation schemes (fixed and dynamic) and provide performance analysis to show that the dynamic vacation scheme gives better temperature and power consumption with acceptable latency overhead.

Then, to understand the effectiveness of the vacation scheme, we develop the thermal, power and latency models for our scheme in the multicore processor by adopting the vacation queuing theory [83, 113]. Unlike single core processor, the thermal behavior of multicore processor is more complicated because of the temperature interaction between cores. Through the use of our models, we can select proper limits for vacation and busy period with a given traffic load.

Lastly, we apply our vacation scheme with the thermal constraint in the multicore processor. With our proposed thermal model, we can properly limit the length of the busy period and avoid thermal emergency. The result shows that, compared with the existing scheme, our vacation scheme can effectively satisfy the thermal constraint while sustaining higher traffic load. Also, through proper vacation period selection, the power consumption can be reduced. Furthermore, by taking the thermal and C-state characteristics of the multicore architecture into account, we propose heterogeneous load distribution, which effectively redistributes the load and heat generation in the spatial domain, and creates more opportunities for power saving under the thermal constraint. To the best of our knowledge, we are the first to develop a vacation scheme considering all temperature constraint, power consumption and latency for packet processing on a multicore server.

This chapter makes the following contributions.

- We propose and implement a vacation scheme on top of the idle states (C-states) provided by the OS and observe the packet processing characteristics.
- We derive analytical models for power, thermal and latency characteristics for the vacation scheme based on the vacation queuing theory.
- We explore the effectiveness of our vacation scheme under the thermal constraint.
- We design and implement heterogeneous load distribution scheme, and evaluate our on-line algorithm experimentally.

3.2 Packet Processing on a Multicore Server

This section describes the high-level network application characteristic, and relates it to the power and thermal behavior while processing on a multicore processor.

3.2.1 Packet Processing Characteristics

The “run-to-finish” applications always consume active power while running, and the temperature simply rises to saturation point until it stabilizes. But network packet processing exhibits an ON/OFF pattern [23,77], where the CPU core temperature rises and consumes active power during the ON period, (also known as busy period) when the packet is being processed. During the OFF period when no packet arrives, the CPU still runs and consumes idle power which is less than the active power. The OFF period between packet processing provides the opportunities to do power and thermal management. Also, the length of ON/OFF period is not deterministic, and depends on the random packet arrival

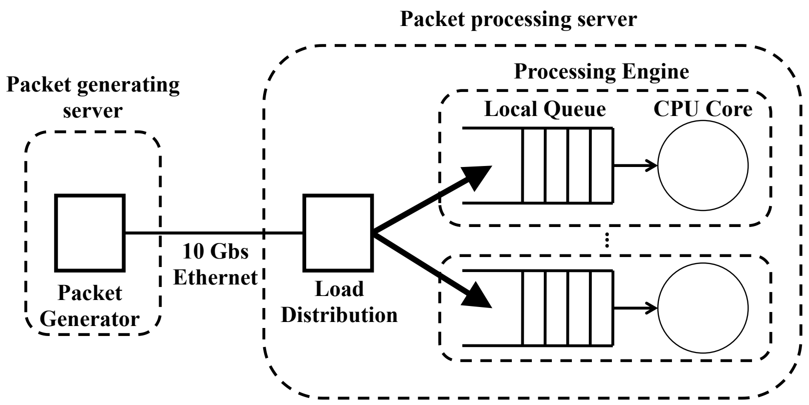
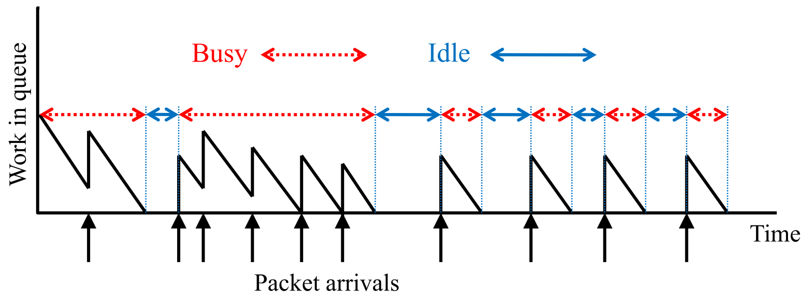


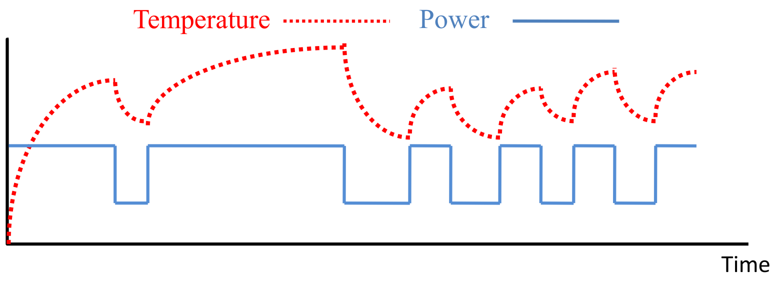
Figure 3.1: The experiment platform

and service rates.

We start by considering how packets arrive and are served. Figure 3.1 shows the packet processing system we use in this chapter; our system consists of a packet generating server and a packet processing server. The system consists of many processing engines (PEs); each PE has a local queue and a CPU core. The processing server receives packets from the packet generator through Ethernet link. Once the packet arrives at the receiving server, it will first go through NIC and TCP stack, and will be passed to the socket. Finally, the application will fetch the packets through *sock_read* system call and process the packets. The incoming packets are distributed among PEs through the load distributor. The packets arriving at each PE are stored in its local queue, and the CPU core constantly checks whether the queue is empty or not. If the queue is not empty, CPU core will fetch and process a packet in the queue in an FIFO fashion driving the CPU to the active state. On the other hand, while the CPU core finds no packet in the queue, it will become idle (entering idle state) until the next packet arrives and is stored in the queue.



(a)



(b)

Figure 3.2: (a) Packets processing pattern. (b) Power consumption and thermal behavior during packet processing.

3.2.2 Power Consumption and Thermal Behavior

Figure 3.2(a) depicts the packet processing pattern of a PE in the multicore packet processing server [31]. The solid lines represent the amount of work in PE queue with time. The solid up arrows under the x-axis represent the packet arrivals to the PE, which cause the step increase of the work in the queue. Although the amount of work could be discrete, without loss of generality, we use slant line representing the packet being processed by a processor with the slope representing the service rate. The busy and idle periods are shown in the top of the figure. Clearly, the busy and idle periods during packet processing are randomly distributed with the expected lengths of $\frac{1}{(\mu-\lambda)}$ and $\frac{1}{\lambda}$, where, λ is the packet arrival rate and μ is the service rate of a PE. Figure 3.2(b) shows power consumption and thermal behavior corresponding to the packet arrival pattern of figure 3.2(a). When PE is active, it consumes high power, and drops to a lower level when idle. The temperature trace is like the shape of sawtooth, which is caused by the continuous busy and idle periods. The longer the busy period, the higher the temperature will rise until a steady state. On the contrary, the longer the idle period, the lower the temperature will fall. Also, the power consumption during the busy and idle periods will decide the steady state temperature. Furthermore, due to the heat flows between CPU cores in a multicore architecture, their temperature behavior will interact with each other. The complicated thermal behaviors pose a major challenge for the temperature management.

3.2.3 Problem with Existing Idle State Power Management Scheme

As we mentioned previously, the CPU core will become idle when it finds no work to do, and still consumes power. Modern CPUs provide succession of per-core idle states (called sleep states or C-states), where different CPU core components are turned off. States are named numerically from C_0 to C_N, where C_0 represents the normal operating mode. As the C-state number increases, the more components in the CPU core are turned off, and the heat generation and power consumption reduces with the increased state transition time. Because of the state transition overhead, it only makes sense to enter a C-state if inactive time is beyond certain threshold, called energy-breakeven-time (EBT), and this threshold increases as the C-state gets deeper.

Because of the EBT, the length of the idle period should be known beforehand to select the most energy efficient C-state a core should enter. In the Linux kernel, the prediction-based sleep-state selection algorithm, called menu idle governor [3], is used for this purpose. It uses a history-based idle length predictor. If the variation among the

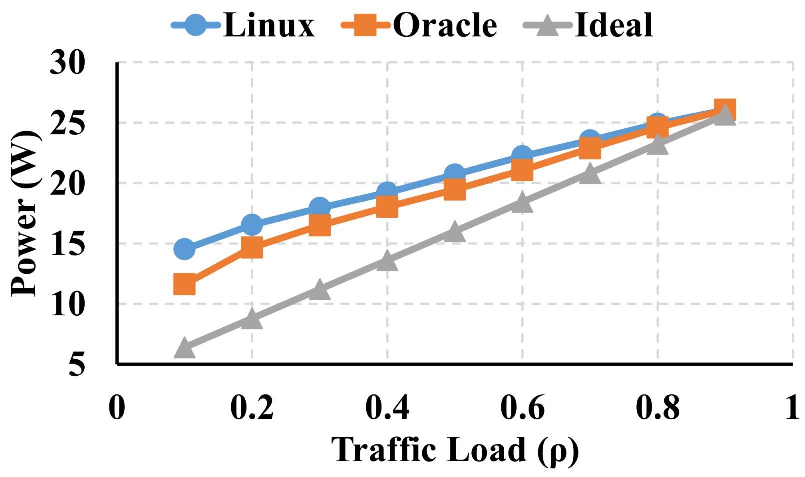


Figure 3.3: Power consumption under different scenarios.

past idle intervals is small, the average of past idle intervals will be used as the prediction; otherwise the next scheduled OS timer event will be used as the prediction. However, due to the *unpredictable nature of the request arrival*, the menu governor or any history based idle period predictor is not very useful with the non-deterministic execution pattern [38], where over-prediction (predicted idle period is shorter than the actual one) will cause energy inefficiencies and under-prediction will cause unnecessary latency increase and power wastage due to the large state transition overhead.

Besides the difficulties for the idle period prediction, another issue for idle power management lies in the *frequent and random* ON/OFF pattern of network and datacenter applications, which greatly affect the effectiveness of using C-states. Fig. 3.3 gives the experimental results of the power consumption of a network application under different traffic loads. There are three scenarios in Fig. 3.3: *"Linux"* represents the power consumption of using Linux prediction-based C-state selection algorithms (menu idle governor), and the results are measured using on-chip energy sensors; *"Oracle"* represents the case when all idle durations are known beforehand so the core can enter the most energy efficient C-state with no prediction error, and *"Ideal"* gives the power consumption with perfect energy proportionality, where all CPU cores go into deepest sleep state during all of its idle time with zero transition overhead. One should notice that the Ideal scheme serves as a theoretical lower bound of the power consumption that can be achieved. In Fig. 3.3, we can see that when the system has highly activity ($\rho > 0.7$), the difference between Linux and Oracle cases are small, because the idle period is mostly short, which make shallowest C-state the most energy efficient. However, due to the inaccurate prediction, the Linux prediction algorithm becomes less effective when the traffic activity decreases, 16% and 25% more power consumption than the Oracle case at = 0.4 and 0.1, respectively. Furthermore, despite the perfect idle state selection of the Oracle case, the existence of short idle periods will prevent the system from going into deep C-state. For this reason, even with low activity, the system operates far from being energy proportional - i.e., consumes power proportionally to its utilization, as shown in the *Ideal* case. Moreover, the high idle power consumption in the existing idle power management scheme also lead to poor thermal behavior, since the degree of cool down during the idle period is dependent on the idle period power consumption.

3.3 Vacation Thermal and Power Management Schemes

In the original system, servers are busy processing packets as long as there are packets in the queue, and become idle when the queue is empty. Although servers spend most of their time idle, conventional power management techniques are unable to exploit these brief idle periods. As a result, the CPU cores still consume high power during these idle periods and generate heat. To address this issue, we propose an approach to reshape the packet processing pattern that creates longer idle period and allows the processors to enter low-power state which consumes less power and reduces heat generation. We call this approach vacation.

In this section, we first propose a vacation scheme with the fixed active period, where the core state changes only when the timer expires. Then, we propose a dynamic vacation scheme, where the state changes dynamically with different events.

3.3.1 Fixed Vacation Scheme

Figure 3.4 shows the flow chart of the vacation approach. We define two states that CPU can be: *working* and *vacation*. In the working state, the core works as in the original scheme, but only goes into the shallowest sleep state (C1) when idle. A working timer, t_{work}, is set when the core enters working state, and the core will transit to vacation state when this timer expires. The purpose of the working timer is to prevent the processor from

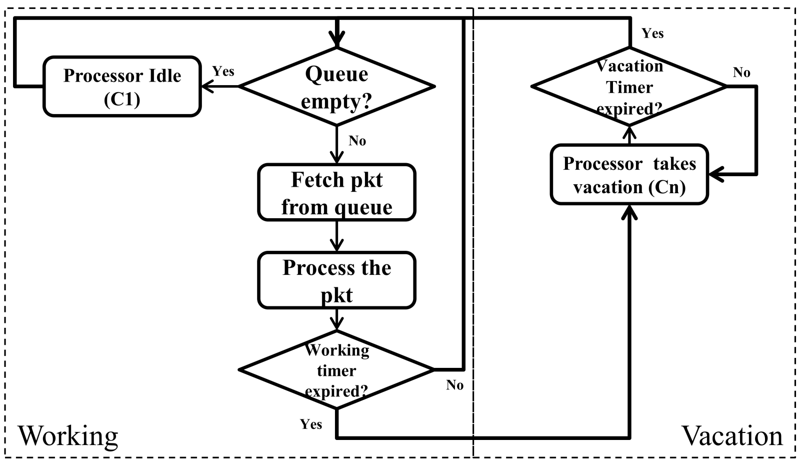
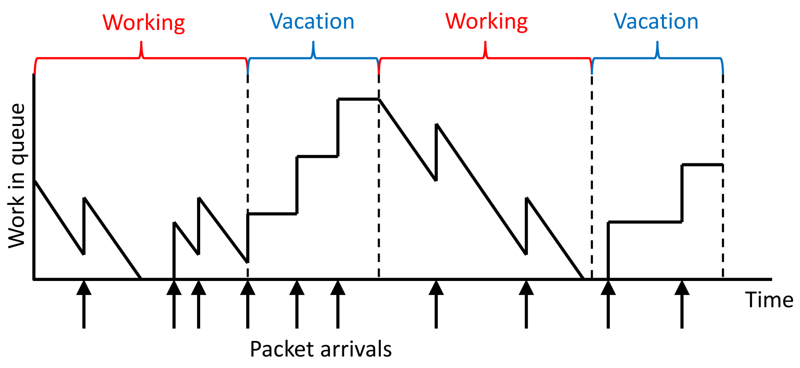


Figure 3.4: Flowchart of the fixed vacation scheme.

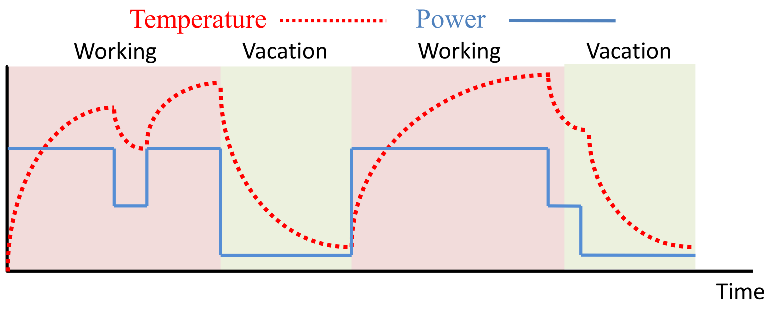
being active for too long and exceeding the given temperature threshold. In the sleep state, the server takes a vacation for time t_{vac}, when the server buffers all the incoming packets in the queue and enters deep sleep state. After t_{vac} expires, the server wakes up and goes back to the working state, and continues serving packets in the queue. The approach is similar to the "buffer and burst approach", described in [100], except that the details of the sleep approach here are different. In buffer and burst approach, the incoming packets are buffered during the buffer period, and these buffered packets are transmitted only in the burst period. We process those packets if they arrive during the working period t_{work}. Also, the buffer and burst approach did not include any thermal consideration like temperature threshold. Finally, we implement the system in real multicore system instead of verifying through simulation [100].

In the fixed vacation scheme, the incoming traffic from processors perspective, is reshaped into short bursts of arrival. CPU cores wake up to process bursts of packets, and then sleep for a long time. The intent is to provide sufficient time for processors to enter the deep sleep states (C3 and C6) with lower power consumption, and faster temperature cool down. Figure 3.5a shows the amount of work in the queue while time advances with the fixed vacation scheme. Compared with figure 3.2a, we can see that the amount of work increases due to the buffering during the vacation. In the working state, instead of many short packet processing time periods, it is now a long busy period.

The effectiveness of the vacation scheme relies on the capability of CPU core to cut down the heat generation and power consumption when it is idle [75]. Although the idle periods between packet processing provides natural thermal modulation, the idle period is



(a)



(b)

Figure 3.5: (a) Packets processing pattern. (b) Power consumption and thermal behavior under fixed vacation scheme.

too short to allow the processor enter deep sleep state in the original system, hence resulting in poor power and thermal efficiency. Vacation alters the distribution of these idle periods, creates longer idle periods and better utilizes the low power sleep state. The power and thermal behaviors are shown in figure 3.5b; where the deeper idle states during vacation decrease the power consumption, and cause the temperature to cool down faster. However, since the incoming packets are pushed into vacation state, there may be higher load on the system during the working period. This leads to increased average power consumption and heat generation during the period. Longer busy period means that the temperature could rise to a higher level, but longer sleep (vacation) state helps lower temperature. Therefore, the efficiency depends on balancing the working and vacation states.

3.3.2 Dynamic Vacation Scheme

In the fixed vacation scheme, there will be cases that the processor will experience short idle time during the working periods, as shown in figure 3.5b. To achieve power saving, we propose a dynamic vacation scheme with the flowchart shown in figure 3.6. The processor switches into vacation state not only when the timer expires, but also when the processor finds the queue empty during the working period. In figure 3.7a, we can see that once the work in queue becomes zero, the processor immediately takes a vacation, and packets accumulate in the queue until the vacation timer expires. Also, since the working period ends when queue is empty, the length of working period becomes non-deterministic and is dependent on the incoming traffic and the length of vacation. However, the working period is always less than or equal to the pre-determined fixed time period, t_{work}.

Figure 3.7b depicts the power and temperature behavior under dynamic vacation;

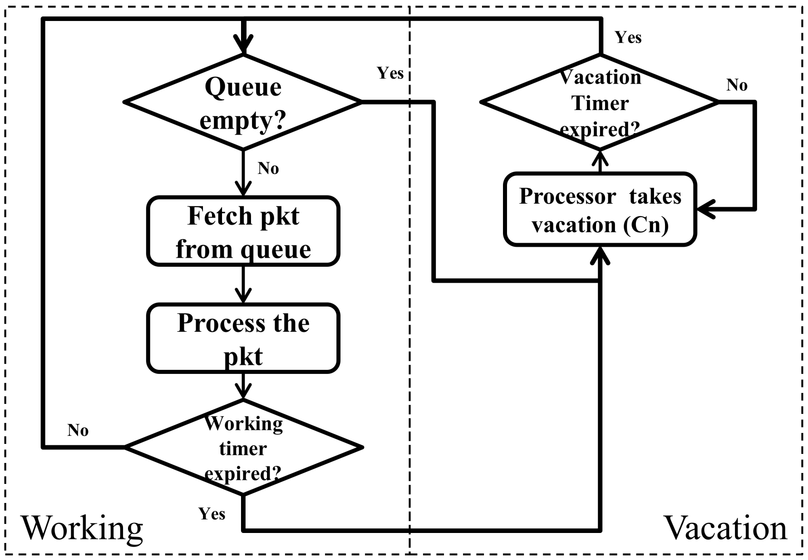
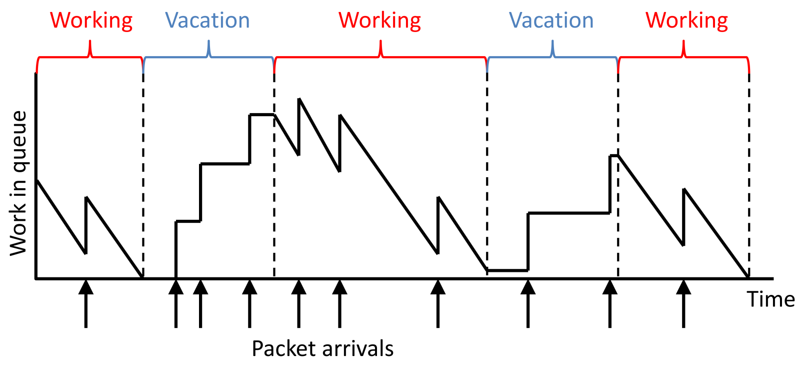
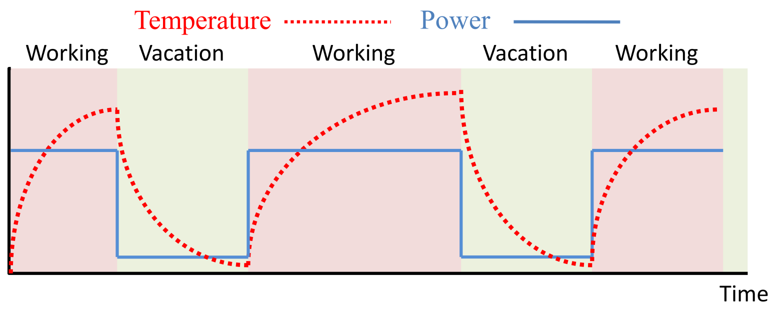


Figure 3.6: Flowchart of the dynamic vacation scheme.



(a)



(b)

Figure 3.7: (a) Packets processing pattern. (b) Power consumption and thermal behavior under dynamic vacation scheme.

we can see that there is no short idle period during the working period, and the power consumption is purely active state power. The thermal behavior is now simpler, without any short cool-down period, the temperature rises until the end of working period, and cools down during the vacation period. In dynamic vacation approach, the length of vacation period becomes more critical to the performance than the original approach. Longer vacation time will reduce the temperature and power consumption. On the other hand, long vacation period will introduce higher packets latency. In the following sections, we show the effectiveness of both vacation approaches with the experimental results, and present the performance models to dynamically determine the vacation length based on the incoming traffic.

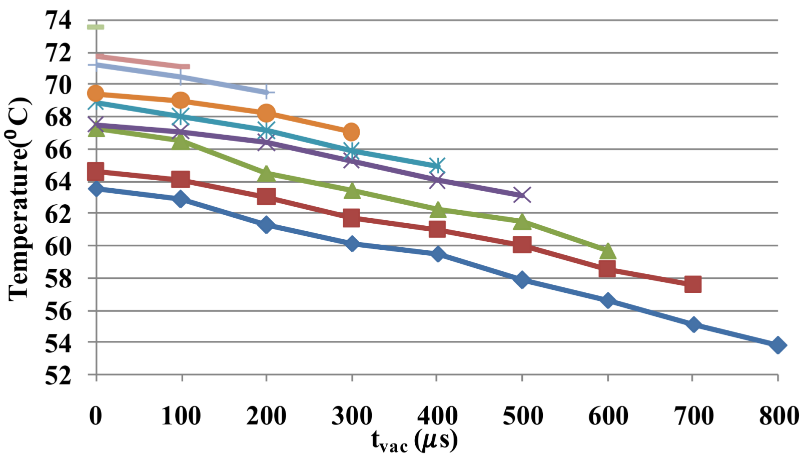


Figure 3.8: Peak temperature under fix vacation approach.

3.3.3 Vacation Schemes Performance analysis

In this section, we present detailed performance analysis for our proposed vacation schemes with real machine experimental results. The experiment setup are given in section 3.4.2.

Fixed vacation scheme

In the fixed vacation approach, design parameters which will affect the packet processing performance are the length of working period (t_{work}) and vacation period (t_{vac}). Let t_{cycle} be the cycle time period, defined as $t_{cycle} = t_{work} + t_{vac}$. For stability of the system, we need to satisfy the condition that $\frac{t_{vac}}{t_{cycle}} < 1 - \rho$, where ρ is the incoming traffic load, defined as $\frac{\lambda}{\mu}$, where λ is the packet arrival rate and μ is the service rate.

Figure 3.8-3.10 shows the experimental results of CRC from NetBench [95] with the vacation approach. In these experiments, we fix the cycle time to 1ms (t_{cycle} = 1ms), and vary t_{vac} from 0 to 800 μsec, where t_{vac} = 0 represents the default scheme. In figure 3.8, we can see that the temperature decreases as t_{vac} increases. Although the busy

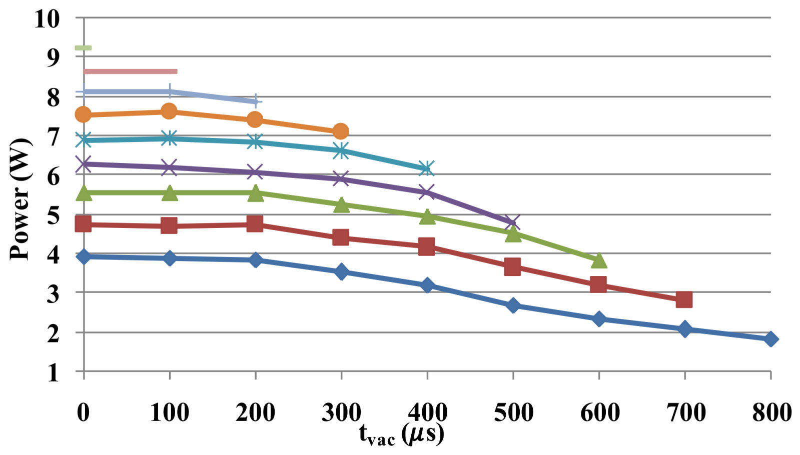


Figure 3.9: Power consumption under fix vacation approach.

period is prolonged due to forced vacation, which causes the longer heat up duration, the temperature cool-down brought from continuous idle period and deeper idle state is more significant. Also, when ρ increases, load during the working period increases, which causes the temperature to increase.

The power consumption results are given in figure 3.9. We can see that it achieves more than 20% of power saving under low traffic load (ρ <0.4), and 50% when ρ=0.1. However, the effect of power saving is diminished when traffic load is high. The reason is that under low traffic load, there exists a large amount of short idle periods. These short idle periods are packed into a long idle period in the vacation state, and allow the CPU core to have enough time to enter deeper C-state. On the other hand, under high traffic load, the idle period is not long enough because working period increases. We can also observe this through t_{vac}. When t_{vac} is smaller than 200 μsec, we can see that the power saving is insignificant, because the short t_{vac} will stop the CPU core from entering deep C-states (C3 and C6).

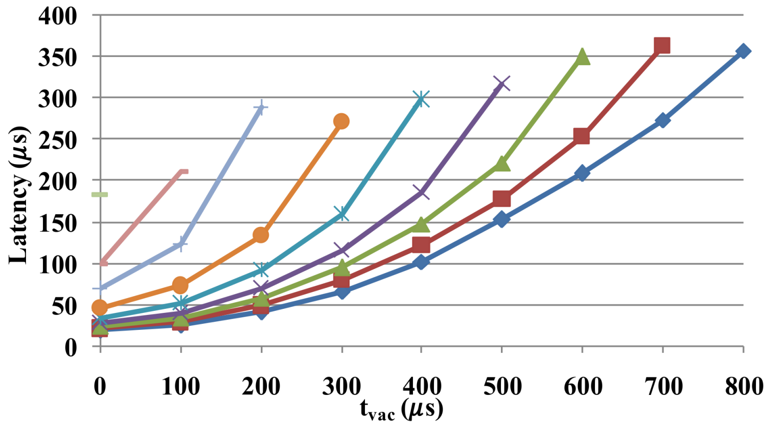


Figure 3.10: Average packet latency under fix vacation approach.

The down side of the vacation approach is that it will increase the overall packet latency. Figure 3.10 shows that the latency increases quickly as t_{vac} increases. There are two factors that will cause this latency to increase. First, when t_{vac} is larger, which means longer vacation periods, the packets arriving during vacation will have to wait for longer time for processing. Second, since all packets arriving during the vacation period are queued, the packets arriving during working period will have to wait for the packets already in the queue to be processed. The longer the vacation period, the more packets are in the queue, thus longer is the waiting period.

To further show that vacation approach achieves power saving through the use of C-states, we record the percentage of the idle time that the CPU core resides in different C-states under low traffic load. In figure 3.11, we can see that the CPU core spends more than 90% of the idle time in C1 state, the least power saving mode, in the default scheme. When t_{vac} increases to 200μsec, the core starts to enter C3 state. When t_{vac} further increases beyond 500 μsec, it enters C6 state and achieves higher power saving because there is long

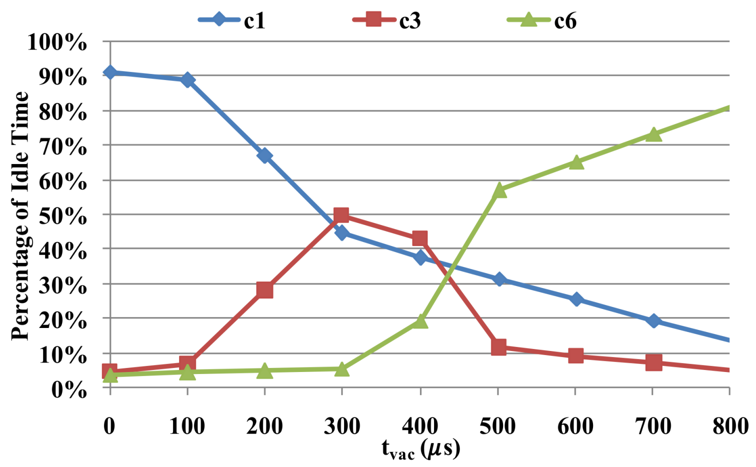
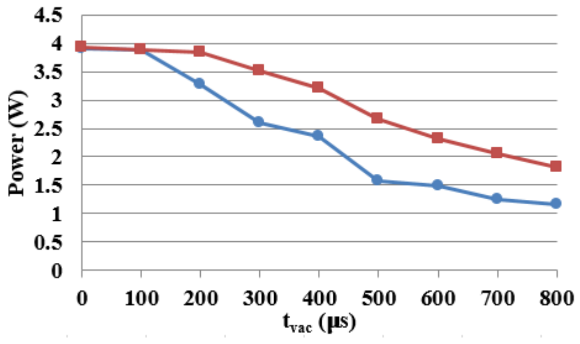


Figure 3.11: Percentage of idle period that the core resides in different C-states under fix vacation approach ($\rho = 0.1$).

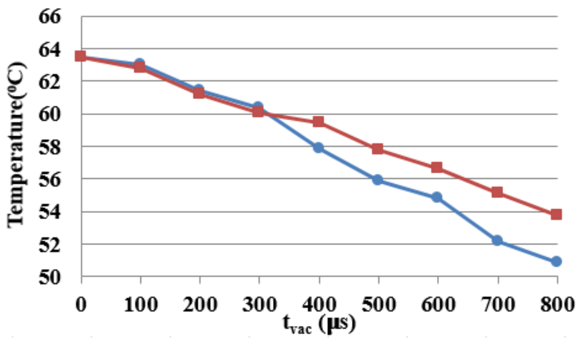
enough idle time. The slope of the power consumption in figure 3.9 increases between t_{vac} = 400 and 500 μsec, since the core spends majority of idle time in C6 states.

Dynamic vacation scheme

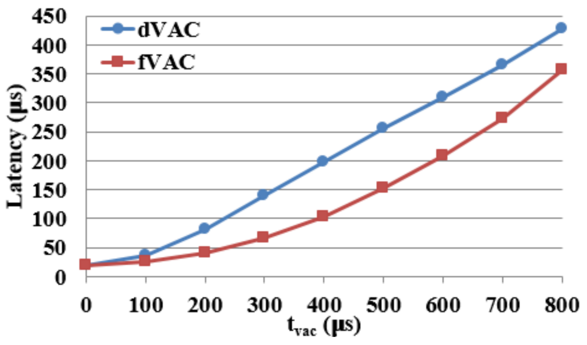
In the dynamic vacation scheme, the CPU core will start vacation as soon as the queue becomes empty. Since there is no more packet to be processed we forcibly end the working period, so we should be able to observe more power saving. The comparison between fixed vacation and dynamic vacation scheme are shown in figure 3.12. Figure 3.12a shows the experimental result of power consumption with different vacation time for dynamic vacation scheme. We can see that there exists significant power consumption drops when t_{vac} reaches 200μsec, corresponding to the processor entering C3 state in the vacation period. The power consumption further drops when t_{vac} reaches 400μsec, when the core starts to enter C6 state. Also, the dynamic vacation approach can achieve higher power saving because there is no idle time during the working period, which may lead to shallow



(a) Power consumption



(b) Peak temperature



(c) Average latency

Figure 3.12: Performance comparison between dynamic vacation (blue line) and fixed vacation approach (red line).

C-states.

To confirm the power consumption is actually coming from the use of deeper C-state, the percentage of idle time that the processor spent in each C-state is shown in figure 3.13. We can see that the C3 and C6 residencies increase at t_{vac}= 200 and 400μsec, which confirms our statement for the power consumption. Also, compared with figure 3.11, we can see that the C1 residency is less than 5% when t_{vac} is greater than 300μsec, which suggests that the dynamic vacation approach indeed packed all the idle periods into vacation period, and effectively uses the deep C state for power saving.

The temperature reduction results are shown in figure 3.12b. We can see that the

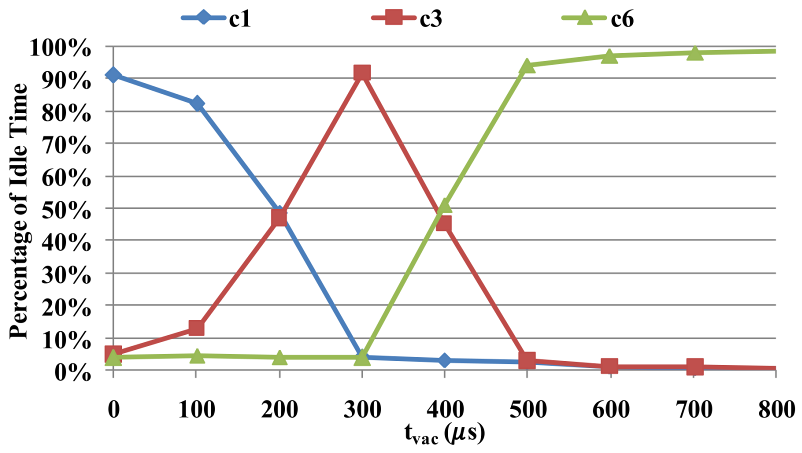


Figure 3.13: Percentage of idle period that the core resides in different C-states under dynamic vacation approach ($\rho = 0.1$).

dynamic vacation approach achieves better temperature reduction than the fixed scheme, especially when t_{vac} is large (2°C more temperature reduction when $t_{vac} = 800\mu$sec). It's because at high t_{vac}, the working period is significantly shorter, 200μsec in fixed vacation and 90μsec in dynamic vacation, but with less than 25% power consumption increasing, which is the source of heat generation. When t_{vac} is smaller, there is shorter heat up duration but higher heat generation, so the temperature reduction of both vacation schemes are similar.

Although dynamic vacation outperforms the fixed vacation in both power saving and temperature reduction, it suffers from higher latency. Figure 3.12c shows the latency comparison between two approaches. When ρ is low, the working period is shortened, the possibility that the packet arrives during vacation period is increased, and those packets will have to wait for the vacation to end, which causes higher latency.

3.4 Multicore Vacation Scheme

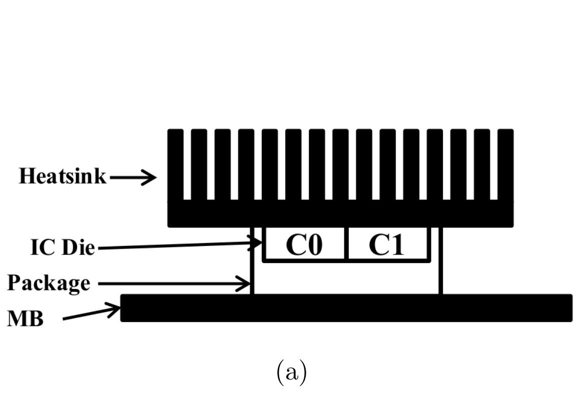
In the previous section, we showed that the dynamic vacation approach is better than the fixed vacation in terms of thermal and power consumption for a single CPU core. In this section, we provide the performance analysis and build the model for the dynamic vacation scheme under multicore architecture. The model is much more complex than the single core model because it considers heat flow between the neighboring cores. As shown in figure 3.1, each CPU core has a buffer and operates independently of other cores. Each core goes on a vacation (idle state) for a fixed period and then wakes up to serve the packets in its queue. We derive the delay model by following a multiple M/G/1 vacation queuing theory. Then, we derive the thermal, power and latency behavior under multicore architecture by applying multicore thermal model and vacation queuing server model. The analytical results are verified through actual experiment in a multicore server.

3.4.1 Performance Analysis of the Vacation Scheme in multicore CPU

In this subsection we propose power, latency and thermal models for the vacation scheme under multicore architecture.

Temperature Model

To understand the multicore thermal behavior, first we need to know how the heat generated by the core flows. Figure 3.14(a) shows the overview of a two-core CPU package. It is composed of a heatsink, which helps heat to dissipate into the surrounding air, and a CPU package, which protects the two-core IC die and sitting on top of the mother board (MB). When the processor is active, both C0 and C1 core will generate heat. The heat will



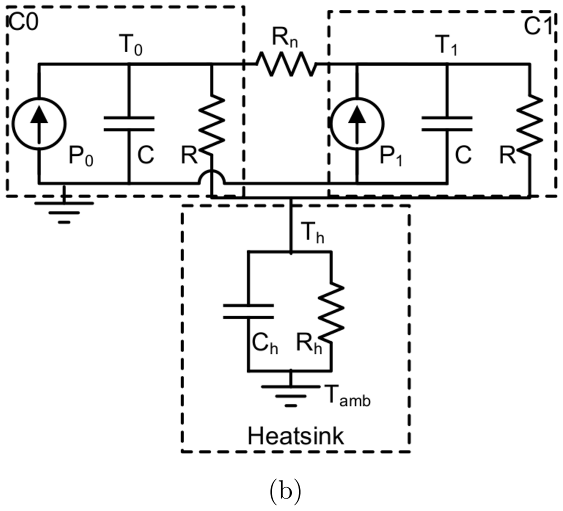


Figure 3.14: (a) The CPU package overview and (b) the extended RC model of a two-core processor.

flow in two directions: vertically, from core to heat sink, and horizontally, from one core to the other. These heat flows make the thermal behavior of the core not being independent among cores in the multicore architecture.

We develop a thermal model by extending the well-known single core RC model [110], and is shown in figure 3.14(b)[1]. The core can be modeled as a current source connected with a pair of resistance and capacitance in parallel. Let $V^i(t)$ and $K^i(t)$ be the temperature in the vacation and working period at time t. By using superposition properties of the circuits, and assuming that the thermal time constant of the horizontal heat flow is the same as the vertical one, we can have the thermal behavior of a core as:

$$\begin{gathered} V^i(t) = \left[(1-K)P_C^i + KP^n\right] R(1 - e^{-\frac{t}{\alpha}}) + V^i(0)e^{-\frac{t}{\alpha}} \\ , K = \frac{R}{R_n + 2R} \end{gathered} \tag{3.1}$$

[1]The thermal model derivation are based on two cores system in this section. However, the thermal model we apply throughout this paper is derived with the same technique based on four core system.

, substituting P_c with P_{work}, we will have the working state thermal behavior $K^i(t)$. In equation (3.1), the superscript i and n represent the modeled core and its neighbor core, P^n is the average power consumption of the neighbor core and $\alpha = RC$ is the thermal time constant. Although the temperature will not converge to steady state because of the continuous state changes, we could derive the thermal behavior in the equilibrium state, which means $V^i(t_{vac}) = K^i(0)$, and $V^i(0) = K^i(t_{work})$. The peak temperature, T_p^i, can be derived as:

$$\begin{gathered} T_p^i = R\left[(1-K)P_C^i + KP^n + (1-K)(P_{work}^i - P_C^i)\frac{1-A}{1-AB})\right] + P_{pkg}R_h \\ , A = e^{-\frac{t_{work}+t_{trans}}{\alpha}} and B = e^{-\frac{t_{vac}-t_{trans}}{\alpha}} \end{gathered} \tag{3.2}$$

, where P_{pkg} is the total power consumption within the CPU package, t_{work} is the length of working period, and t_{trans} is the C-state transition time. The last term is the temperature of the heat sink and the rest depict the temperature rising of the core.

Power Model

Since the vacation scheme only reshapes the packet processing pattern, it does not introduce extra idle/busy time. The average power consumption of a core under vacation scheme is:

$$P_{vac\_avg} = \frac{P_{work} \times t_{work} + P_C \times (t_{vac} - t_{trans}) + P_{max} \times t_{trans}}{t_{work} + t_{vac}} \tag{3.3}$$

, where P_{work} is the busy period power consumption, t_{work} is the length of busy period, t_{tran} is the C-state transition time, and P_{max} is state transition power consumption. Since the power consumption are independent among cores, the overall system power consumption will simply be the summation of that of each core.

Latency Model

Since the working period will either end when the queue becomes empty or timer expires, the system is equivalent to a M/G/1 queue with vacation and non-gated service [83, 113]. We use mean value analysis to analyze the packet latency in vacation approach. We assume that the working period time limit is large enough that the latency of packet being deferred to the next working period can be ignored. Under this assumption, we can derive the average latency as:

$$E(L_{avg}) = L + O + \frac{t_{vac}}{2} \tag{3.4}$$

, where L is the latency without applying our technique, O is the additional latency caused by packets which are processed in the next working period. The last term is the additional latency of the packets arriving during the vacation.

Since every core processes packets and takes vacation independently, the latency will be independent among cores. As a result, the overall latency is the weighted sum of the latency of every core, where the weight will be the ratio of the load for the core to the overall incoming load.

3.4.2 Experiment Setup

As shown in figure 3.1, we use two multicore machines, one packet generating server and one packet processing server. The packets are generated and sent over the Ethernet to the processing server. We are mainly interested in the behavior of the packet processing server. The packet processing server is an Intel i7-2770 quad-core processor. The Intel Hyperthreading, Turbo boost, and Enhanced C1 state (C1E) are disabled through BIOS setting and the machine specific register (MSR). The CPU fan speed is set to be fixed at 1000 RPM. The server is equipped with on chip energy sensors for all four cores, and per-core temperature sensors. Both energy and temperature readings are collected from accessing MSR at the rate of 1000 reading per second. The temperature readings from the MSR are rounded to the integer.

Our experiments in this paper are based on real world traffic traces from CAIDA [2]. The traces are of 60 minutes long duration, captured on JUN 19th, 2014, containing the timestamp and packets header. The network traces are scaled and replayed by the packet generating server with different traffic load to our packet processing server. Since our proposed technique is only depending on the incoming load and the service power consumption, the performance of our scheme does not rely on the application characteristics. We choose *CRC* [95] as the packet processing application, which is executed in a multi-threaded fashion with packet-level parallelism. The service time of CRC is linear to the packet size, and is 10 sec in average. The power consumption during packet processing is 6.5W per core. The maximum temperature rise is 60°C, which is the difference between maximum temperature and idle temperature. The maximum temperature is the steady-state temperature when there is no idle time within the entire packet processing duration.

3.4.3 Performance evaluation of the multicore vacation scheme

With the multicore performance model, we now discuss the effectiveness of vacation scheme in multicore architecture. Figure 3.15 - 3.17 gives the measurement results of three performance metrics, which are temperature rise, power consumption and latency. Also, they are compared with the results obtained through our model. We do not show the power and latency model results because they are indistinguishable to the measurement results. In this experiment, we vary $tvac$ from 0 to 1000 μsec (x-axis), where $tvac = 0$ represents the default scenario without applying our approach. Also, we select three different incoming traffics, $\rho = 0.1$, 0.4, and 0.7, representing low, medium and high traffic. Furthermore, the incoming traffic is evenly distributed among cores. The detailed discussions are presented below.

Temperature Rise

The vacation scheme can help reduce temperature in core and package levels. In core level, the vacation period provides a longer idle period for the core to cool down compared with the default scheme. Also, when the core enters deeper C-states, the core will cool down to a lower temperature, as we discussed in the previous section. In package level, the reduced power brought from entering deep C-states will reduce the total package power consumption. As the result, the heatsink temperature will decrease. Figure 3.15 shows the measured temperature rise of a core, defined as the difference between the peak core temperature and the temperature when the system is idle. One should notice that the on-chip thermal sensors only report temperature reading rounded to the integer. Also, we choose the vacation period length no more than 1 ms because of the possibility that the background system service will wake up the core during the vacation is high when longer vacation period is chosen.

First, it is clear that the temperature rise is reduced significantly with the intro-

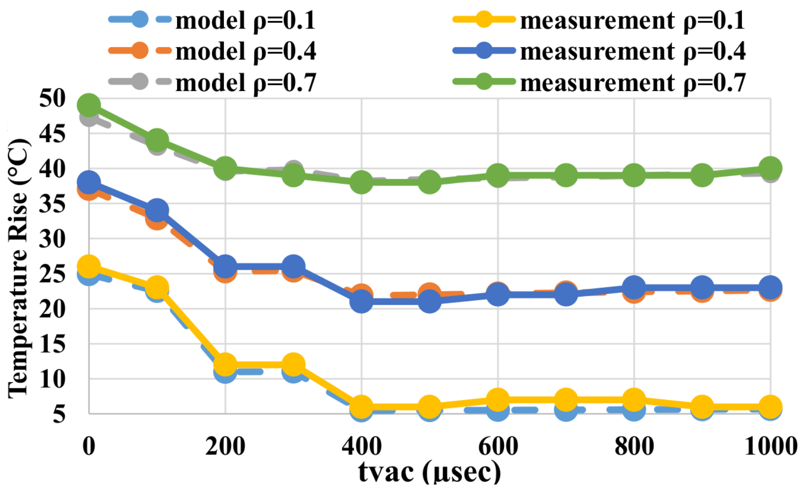


Figure 3.15: Modeled and measured temperature rising of the vacation scheme under different traffic.

duction of the vacation period. In the vacation scheme, the core will enter the corresponding C-state by comparing the vacation period length to the EBTs. At short *tvac*, the core will enter C1 state during vacation. As *tvac* increased beyond 200 and 400 μsec, the core will enter C3 and C6 states instead, hence help the core cool down. Our scheme can achieve maximum 20, 16 and 10°C peak temperature reduction at low, medium and high traffic. Also, we can see that there exists an optimum vacation period length so that the peak temperature rise is minimized. When the vacation length is shorter than this point, because of the state transition overhead, the cool-down during the vacation period is limited. On the other hand, when the vacation length is longer than this point, the prolonged working period will result in high temperature rise. This effect is significant especially when load is high because of the longer working period.

Figure 3.15 also gives the modeled temperature rise based on equation (3.1) and (3.2), and serves as the model verification of our proposed thermal model. In our experiment

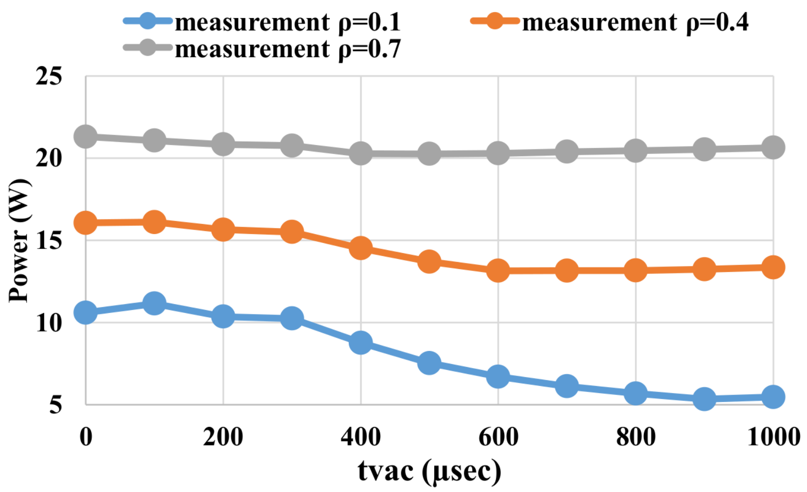


Figure 3.16: Measured power of the vacation scheme under different traffic.

platform, the value for K in equation (3.1) and (3.2) is set to be 0.18. Also, the working period length is chosen to satisfy the stability constraint, $t_{work}/(t_{work} + t_{vac}) \geq \rho$. The result shows that the average difference between modeled and measured temperature is less than 0.5°C, thus our proposed thermal model can accurately describe the core thermal behavior.

Power Consumption

The power consumption results are given in figure 3.16. The default power consumption is when $tvac = 0$. We can see that it achieves 50% power saving under low traffic load. However, the power saving decreases to 20% and 5% when traffic load is medium and high. It is because that the power saving is achieved by entering deeper C-state during the vacation period. When the load is low, the ratio of the vacation to working period is high, thus significant amount of power can be reduced; On the other hand, this ratio decreases with increasing traffic load, as a result, the power saving decreases.

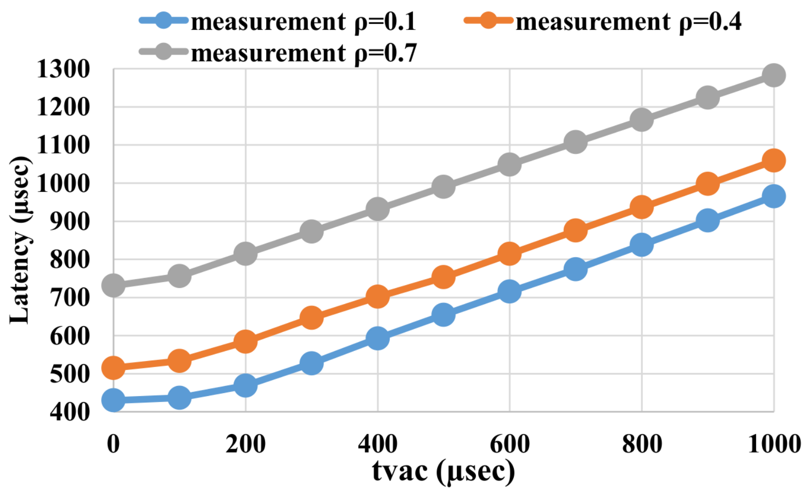


Figure 3.17: Measured latency rising of the vacation scheme under different traffic.

Latency

The down side of introducing forced vacation period is that it will increase the average packet latency. Figure 3.17 shows that the average packet latency increases almost linearly as *tvac* increases, which is predicted by equation (3.4). There are two factors that will cause this latency to increase. First, when *tvac* is larger, the packets arriving during vacation will have to wait for longer time for processing. Second, due to the limited working period, there could be some packets left in the queue by the end of the working period. These packets will introduce some additional latency because they have to wait for an entire vacation period. Given a latency constraint, the vacation period can be determined at a particular load.

3.5 Multicore Vacation Scheme with Thermal Constraint

In this section, we will discuss the performance of the vacation scheme under thermal constraint.

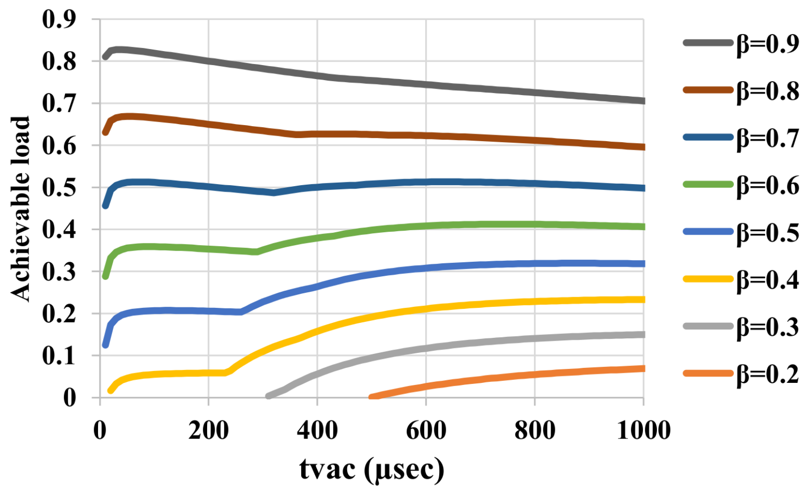


Figure 3.18: Achievable load with vacation scheme with different thermal constraint.

3.5.1 Achievable throughput

Our focus is on keeping the peak temperature below the thermal constraint, T_c, instead of using working and vacation period length as the input in equation (3.2). By rearranging the equation, we can use thermal constraint and the vacation period as the input variables, and get the limit of the working period length, **BP_limit**. This limit represent the maximum time period for the working period without peak temperature exceeding the thermal constraint. Moreover, the limit of working period means that the service capability of a core is limited. In other word, the achievable load that a core can serve without overflowing the queue is limited. We use the term achievable load and sustainable load interchangeably in this section. The achievable load is formulated as:

$$\rho_{achieve} = \frac{BP\_limit}{BP\_limit + t_{vac}} \tag{3.5}$$

Figure 3.18 plots the achievable load under different thermal constraints. The thermal constraint T_c is defined as idle temperature plus a parameter, β, times the maximum temperature rise. In this figure, we can clearly see that as the thermal constraint increases, the sustainable load increases as well. In general, to achieve maximum sustainable load, the *tvac* should always be chosen long enough so that a core can enter C6 state. However, when the β increases beyond 0.7, the *tvac* which achieved maximum sustainable load becomes lower, which means that instead of C6, a core should enter C1 state during the vacation to maximize the service capability.

Figure 3.19 gives a detailed example and can help in understanding the behavior of the sustainable load under vacation scheme. In this figure, we plot the achievable load with thermal constraint set to be 70% of the maximum temperature rising, which corresponds to $\beta = 0.7$. The x-axis is the vacation period length, varying from 10 to 1000 μsec. We separately plot C1, C3 and C6 curves representing the C-state where the core enters during the vacation period. In this figure, we can see that the sustainable load will peak at two points, one is at $tvac \approx 100\mu$sec with C1 state, and the other is at $tvac \approx 600\mu$sec with C6 state. The reason behind the C6-peak is that the heat generated during the vacation period is low. Thus, even though it requires more transition time to enter C6-state, the peak temperature will be reduced and higher load can be sustained. On the other hand, although entering C1 state during the vacation does not reduce much heat generation and peak temperature, because of the low transition time, it does not spend much time and power on state transition. As a result, it can sustain higher load under the thermal constraint. The dotted curve (red) plots the sustainable load with each *tvac* value, and it corresponds to the $\beta = 0.7$ curve in figure 3.18.

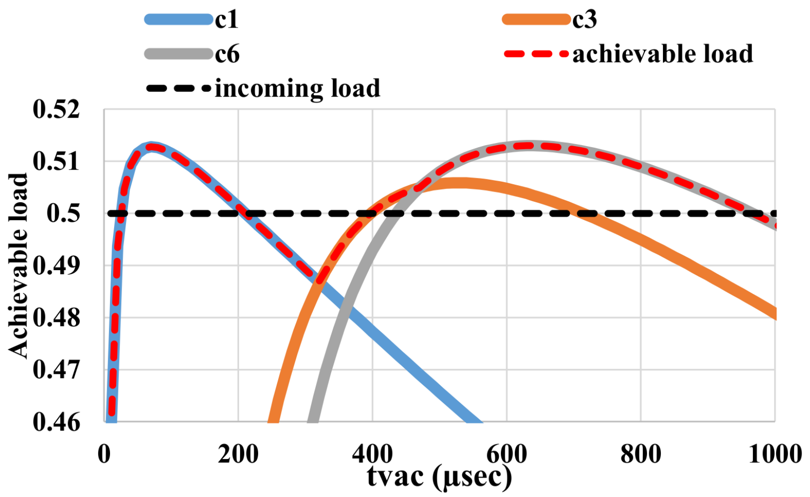


Figure 3.19: Achievable load with vacation scheme under thermal constraint with β=0.7.

When the incoming traffic is given, the *tvac* can be chosen as any point between the intersections of the load line and the C-state curve. For example, in figure 3.19, the dotted line (black) represents the incoming traffic, $\rho = 0.5$. There are three regions where the *tvac* could be: (1) C1-region between 60 and 200 μsec, (2) C3-region between 400 and 700 μsec and (3) C6-region between 430 and 940 μsec. Since every point within these regions can satisfy the incoming load, choosing the exact value for *tvac* depends on other performance metrics, such as minimizing power consumption or latency. Achieving low latency while satisfying thermal constraint is straightforward since the latency is linearly increased with the vacation period. By choosing the lowest *tvac* within the above mentioned regions, we could have the lowest latency and satisfying thermal constraint.

To achieve low power while satisfying thermal constraint, we should choose *tvac* in the C6-region. By applying the C6-region range to equation (3.3) and substituting t_{work} with BP_limit, we can find the configuration with lowest power consumption and satisfying thermal constraint.

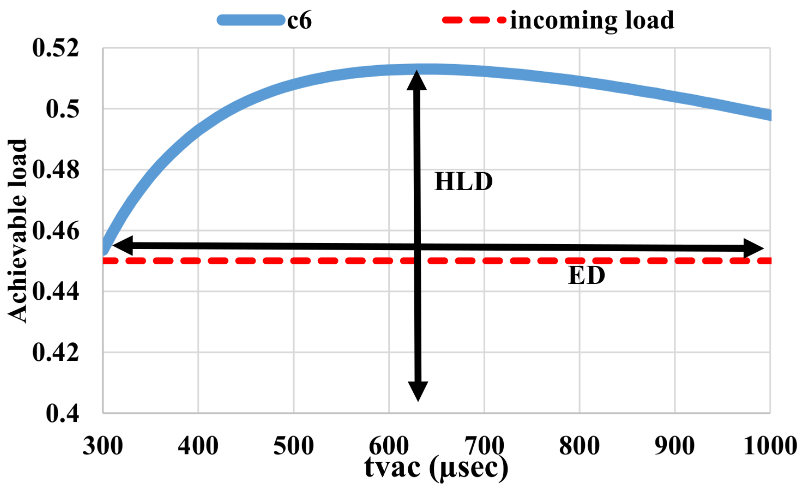


Figure 3.20: Heterogeneous load distribution illustration.

3.5.2 Heterogeneous Load Distribution

Until this point, we assumed that the incoming traffic is equally distributed (ED) among cores. However, this might not yield to the optimum power or latency considering multicore thermal behavior and power consumption. By giving each core different amount of load, the performance under thermal constraint could be improved. In this section, we propose our heterogeneous load distribution technique (HLD) and give an example about how HLD can improve the performance.

Figure 3.20 plots the C6-region with $\beta = 0.7$ and incoming traffic $\rho = 0.45$, and is used to illustrate how HLD works. Equally distribution (ED) selects the *tvac* with the lowest power consumption, which is equivalent to choosing a point on the load line (dotted) in the x-direction. Since every point below the C6-curve and above the load line satisfies the thermal constraint and traffic load, we can give different traffic load to each core. This HLD is equivalent to choosing a point in y-direction, as long as the sum of the traffic load

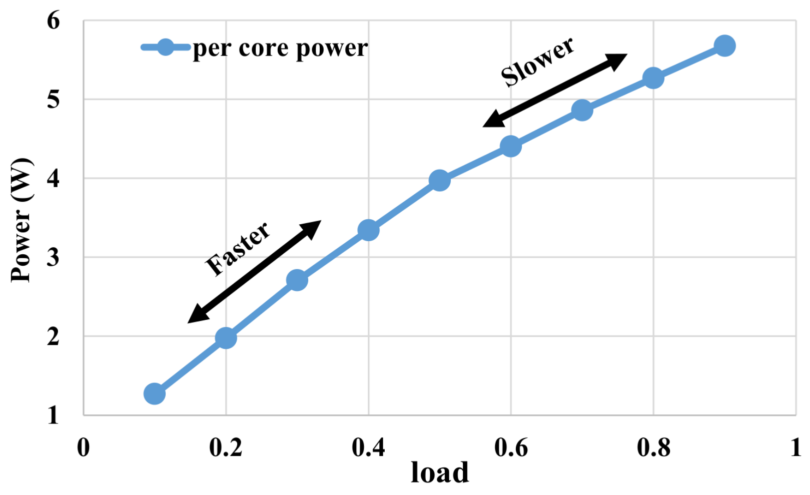


Figure 3.21: Per-core power consumption with different traffic load under ED.

of each core is equal to the total incoming load.

The effectiveness of the HLD comes from the fact that the lowest power consumption of a core achieved by the vacation scheme does not increase linearly with the increasing load. Figure 3.21 gives the per-core power consumption with different incoming load under vacation scheme. We can see that the power-load curve is superlinear, which means that the power increase/decrease is faster at lower load compared with higher load. Hence, among a pair of cores, the total power of these two cores could be reduced by migrating load from one core to another.

We develop a heuristic which chooses the load distribution for minimizing power consumption by the following steps: (1) pick two neighboring cores that haven't been selected before (2) compute the maximum sustainable load. (2) migrate the load from one core to the other so that one core will have the maximum sustainable load, and (3) compute the new power consumption. If it's less than previous power consumption by a threshold Δ, we commit to this load migration and go back to step (2), otherwise we go back to step (1).

In our heuristic, we can see that there is a loop between steps (2) and (3). The reason is that after the load redistribution, the sustainable load for the higher load core will increase because of the decreased average power of its neighboring core, and further redistribution may give better power saving. The threshold Δ is a design choice and we find that 2% of the original power consumption is a good choice to balance between computation time and power saving.

3.5.3 Performance evaluation of HLD

In this section, we present the experimental results and provide some insight on the effectiveness of the HLD in power saving under thermal constraint. First, we want to see how much load our HLD vacation scheme can sustain with different thermal constraints. Since there is no existing work considering thermal constrained packet processing on the multicore system, we compare our scheme with the state-of-the-practice thermal management scheme implemented in the Linux kernel [11]. In this scheme, the core temperature is monitored, and the core frequency is decreased when the temperature exceeds the thermal threshold, and is increased when it falls below the threshold.

Figure 3.22 shows the result of achievable load for ED, HLD and default Linux thermal management scheme under different thermal constraints (β). We can see that our proposed schemes outperform the Linux scheme in three aspects: (1) since the Linux scheme reactively responds to monitored temperature, it can not maintain the core temperature under the thermal constraint. (2) under a given thermal constraint, our scheme allows higher load to be processed by the cores, and (3) our scheme can still perform packet processing

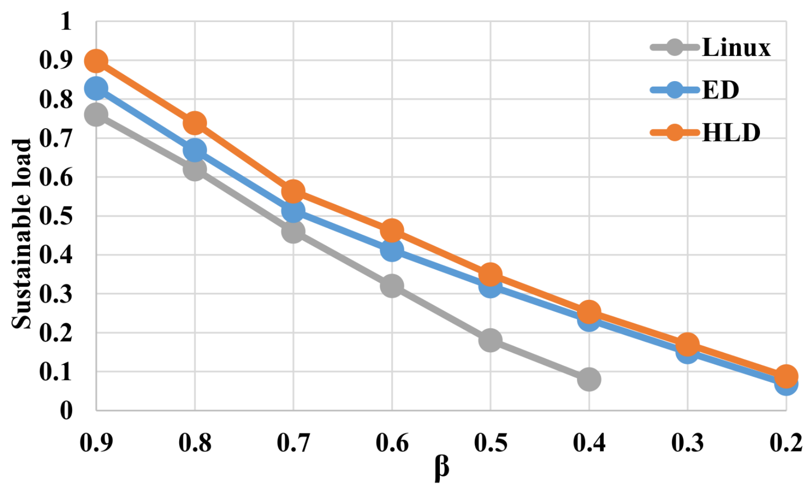


Figure 3.22: Sustainable load of different thermal management scheme.

with lower thermal constraint. Also, because of the load and heat redistribution, HLD can sustain 8% more load with the same thermal constraint compared with the ED scheme at high β, and 5% more load in average.

The power consumption of ED and HLD vacation scheme are given in Table 3.1. In this table, the percentage number in each cell are the power consumption of each scheme normalized to the Linux scheme power consumption. The region that Linux scheme cannot satisfy are marked as N/A. We can see that both ED and HLD vacation schemes can achieve power saving over the Linux scheme. When the load is low, we can see that ED can achieve

	$\beta = 0.9$		$\beta = 0.8$		$\beta = 0.7$	
load	ED	HLD	ED	HLD	ED	HLD
0.2	84.1%	72.7%	84.3%	73.5%	84.0%	75.1%
0.3	87.1%	76.4%	87.0%	77.5%	87.5%	79.8%
0.4	91.1%	81.1%	90.3%	82.5%	91.0%	84.5%
0.5	92.9%	88.2%	91.8%	87.3%	N/A	
0.6	94.5%	93.2%	93.1%	92.2%		
0.7	95.1%	95.1%	N/A			

Table 3.1: Normalized power consumption of ED and HLD.

good power saving, 14% in average. On top of that, HLD will save more power, 25% in average, which indicate the effectiveness of power saving under thermal constraint of the HLD vacation scheme. On the other hand, the power saving achieved by both ED and HLD decreases as the load increases. It is because as the load gets closer to the maximum achievable load, for ED, the *tvac* with lowest power will not fall between the C6-region, leading to small power saving. For HLD, since the y-distance is shorter, the range for load migration is smaller, thus less improvement can be achieved.

Chapter 4

Power Management with Smart Sleep

4.1 Introduction

In this chapter, we propose our multicore idle period power management scheme, called *"Smart Sleep"*. Smart sleep is composed of two parts. First, we propose a per-core power management scheme by modifying the vacation scheme we proposed in chapter 3, which dynamically adjusting the per-core sleep time, during which the core can enter deep idle state without being waken up prematurely. Note that we did not limit the length of busy period in smart sleep to alleviate the impact on the performance. Our per-core power management scheme is motivated form buffer and burst [100] that developed forced idle schemes for a network of routers. However, they focus on device level idle and did not discuss the performance impact with various sleep states. Second, we introduce the dynamic core sizing, which turned of a subset of cores completely. Fig. 4.1 shows the state of the cores in a CPU. At a particular instant, some cores are *off*, some are actively processing requests and some are *sleeping*. The difference between *sleep* and *off* lies in whether a core can accept request arrivals. Requests arriving at the *sleeping* core are buffered and their processing is postponed until the end of the sleep period. On the other hand, requests arriving at the *off* core are lost, since the entire processing engine, including the buffer, is off. Hence, an off core should not receive any request before it is turned on. We derive the expected idle and busy periods by considering the length of the core sleep, request service time and the traffic load. By combining these two models, we can find the behavior of power consumption under our proposed scheme. We derive the response time model using the M/G/1 queuing model with vacation [113]. Finally, we validate our model through implementation using real applications and off-the-shelf multicore server. The smart sleep scheme is closest in terms of concept to [48], which focused on developing various policies to determine when and which sleep states the server should enter. However, they focus on the centralized queue scheme where off state is a natural extension of the sleep states. The multicore packet processing is usually modeled as distributed queue scheme where off and sleep states differ significantly. Second, they neither adjust per-core sleep time nor incorporate various C-states in the CPU hence suffers from long transition overhead, thus is ineffective in fast packet processing that falls in the microseconds level.

Although the smart sleep scheme can achieve power saving, there exists some limitations on its effectiveness in the multicore processor. First, the CPU cores will spend full power on state transition (between active and sleep), and this power overhead reduces

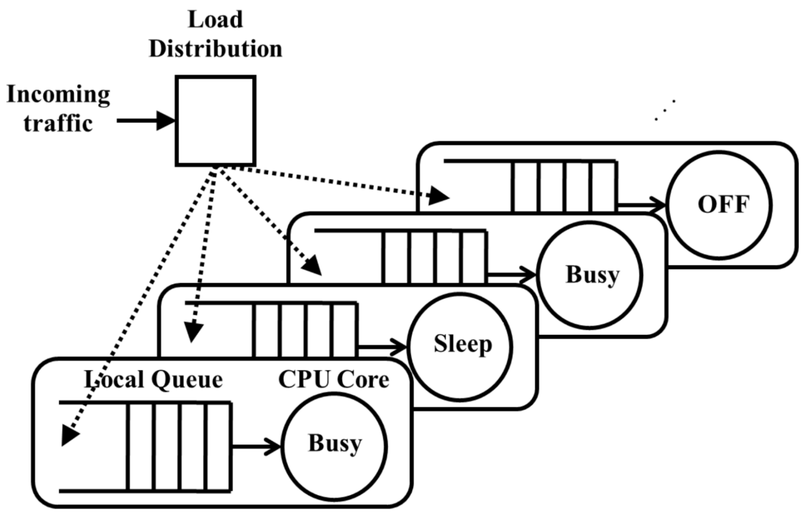


Figure 4.1: Illustration of our proposed multicore smart sleep scheme.

the benefit of the smart sleep. Moreover, depending on the QoS requirement, the length of sleep period will be limited since it will introduce additional latency. To overcome these limitations, we propose dynamic core sizing to maximize the power saving by adjusting the number of active cores on the fly. While the smart sleep scheme saves power through consolidating request processing in the time domain, core sizing serves as another control knob, which utilizes the space domain in the multicore processor. Our idea is to put a subset of cores into off state and process requests with the remaining active cores, where the smart sleep is applied, to further lower the power consumption. Unlike core parking [92], which shuts down the core based on the traffic load and throughput demand, our proposed scheme makes the decision by considering the network traffic pattern, core idle states and latency requirement simultaneously without compromising the throughput and results in a better performance-power ratio.

In this chapter, we consider a multicore architecture combining the concepts of sleep with core sizing [80, 92]. So, minimizing the power consumption becomes a two-

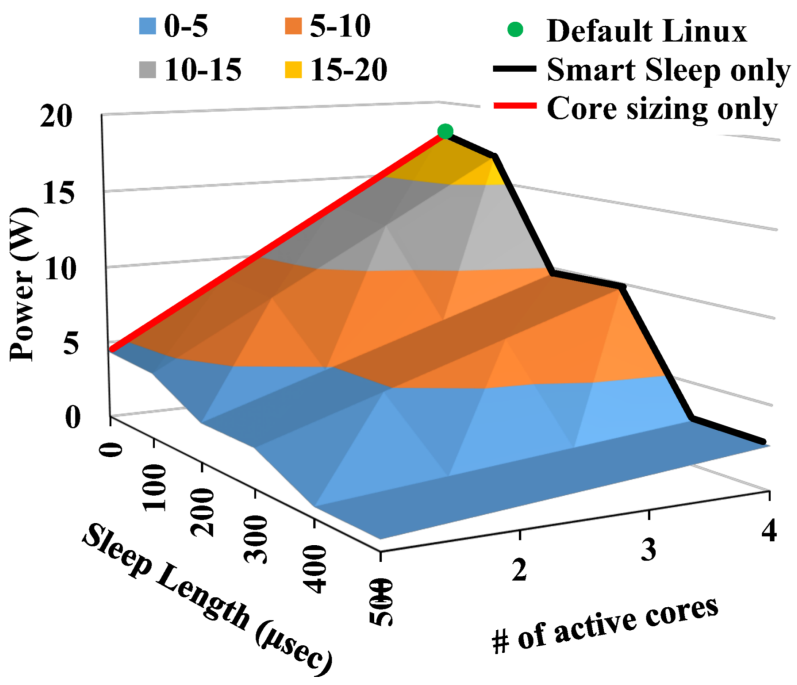


Figure 4.2: Power consumption under default (green dot), smart sleep only (black line), core sizing (red line) and multicore smart sleep scheme (3D surface).

dimensional optimization problem with both time (sleep) and space (off) variations. Fig. 4.2 gives the experimental results for the power consumption under different configurations. We can clearly see that the power consumption will vary depending on both the number of active cores and per-core sleep time for a particular traffic intensity. The power consumption from the previous papers [92, 100] can be represented as a line with single control knob (red and black lines). Furthermore, although not shown in this figure, request latencies behave almost opposite to the power consumption. This gives rise to complex optimization problem of minimizing power while meeting the QoS requirement.

Considering power and latency tradeoffs, we design an algorithm that can intelligently compute the *sleep time* and the *active periods* of cores so that the power consumption is minimized while meeting the required QoS(latency) constraint. We develop a runtime, which monitors the traffic, predicts future traffic demand and then applies different models to arrive at an optimal configuration. We verify our analysis through real experimentation under both synthetic and real world traffic trace with four applications, *memcached, IPv4 routing, URL and IPSEC*, commonly deployed in real-world network and datacenters.

In summary, this chapter makes the following contributions.

- We provide a detailed analysis of processing characteristics of request-based applications and per core sleep states (C-states) using measurements on real systems.
- We build a per-core smart sleep scheme to achieve power saving via regulating sleep periods and per core idle states.
- We analytically model the power and average response time characteristics. Through experimental results, we build an empirical model for the response time distribution.
- We explore further power saving opportunities in multicore architecture by applying core sizing with per-core smart sleep.
- We propose a runtime framework, which dynamically adjusts the size of active cores and the length of sleep period based on the predicted traffic pattern.
- We verify our analysis and evaluate our pro-posed schemes by running real network and datacenter applications on a Linux server, and demonstrate their effectiveness.

4.2 Power Consumption in Network and Data Center Applications

This section develops a high-level model for request-based applications, and relates it to the power behavior during multicore execution. Also, we study the default C-states behavior in Linux and relate it to request processing.

4.2.1 Basic Request Processing Behavior

We start our analysis by showing how requests arrive and are served. In network applications, Receive-Side Scaling (RSS) resolves network load processing on a limited single-CPU by enabling a number of available CPUs in a multicore system by introducing multiple receive queues. Compared to the default load-balance scheduling in Linux kernel (with *irqbalance* enabled), RSS balances the workload on connection level rather than per packet basis [9]. Each receive queue will be assigned to a core in the system, which processes all the packets in the same connection. With RSS, the NIC driver provides the ability to schedule receive packets among CPU cores, and ensures that the processing associated with a given connection stays on an assigned CPU core. The NIC implements a hash function and the resulting hash value provides the means to select a CPU core by inserting the packet into the corresponding receive queue. This distributed queuing scheme has several advantages over the centralized scheme. First, the distributed queues prevent the lock contention existing in the centralized queue scheme, which greatly impacts the performance [9, 112]. Second, because packets in the same connection share not only IP layer information in the header, but also some potential common sections in the payload, processing packets in the same connection can improve I/O latency and throughput by preventing additional data movement and maintaining a better cache locality. Similarly, latency critical datacenter applications, such as memcached [7,47], also implement the connection level load balancing with distributed queue scheme by statically building the connection-core mapping during the application initialization phase.

Fig. 4.1 depicts a general view of request based application processing in multicore architecture. In our example distributed queue system, there are many processing engines (PEs), where each PE is composed of a local queue and a CPU core. Incoming requests are distributed among PEs through the load distributor. Requests arriving at each PE are stored in that PE's local queue, and the CPU core will fetch and process a request in the queue in an FIFO fashion. On the other hand, if there are no pending requests in the queue, the core will enter idle state and stay there until the next request arrives, then will become active and process the request. Note that the system can be modeled as multiple M/G/1 queues, instead of a M/G/k queue, since the incoming requests are scheduled and inserted into distributed queues.

4.2.2 Problem Formulation

Our optimization objective is to minimize average power consumption of a multicore processor with N cores, while satisfying latency constraint. Given a set of R requests, we denote the latency of request r by L_r, for $r = 1, 2, \ldots, R$. Thus, we formulate the power minimization problem as follows:

$$\text{minimize} \qquad P^{N}_{processor} \tag{4.1}$$

Table 4.1: Characteristics of CPU core C-states.

Sleep State	Transition time	Energy-Break-Even Time (EBT)	Power
C0	0μsec	0μsec	5-7W
C1	1μsec	1μsec	3.8W
C3	59μsec	156μsec	1.95W
C6	80μsec	300μsec	0W

$$s.t., \qquad L_{avg} = \frac{1}{R}\sum\nolimits_{r=1}^{R} L_r \le L_c. \tag{4.2}$$

where L_c is the latency constraint.

Note that in some scenarios, instead of constraining on the average latency, the QoS constraint will be on tail latency. The tail (95%ile) latency for the set of R requests, denoted by L_{tail}, is the $0.95 \times R$-th largest latency among these R requests. In this scenario, the constraint function in equation 4.2 will be modified into

$$L_{tail} \le L_c. \tag{4.3}$$

4.2.3 Power Consumption Model with Idle States

As we mentioned previously, the CPU core will become idle when it finds no work to do, and still consumes power. Most of the modern CPUs provide succession of per-core idle states (also called C-states) to save power. In our experimental platform with Intel Quad-core processor, there are three CPU idle power states, as shown in Table 4.1, which are achieved by disabling different components in the CPU core. The C-states are numbered starting at C0, which is the normal CPU operating mode, and the core is fully turned on. The higher the C-state is numbered, the lower the power a CPU core will consume. In the state C1, part of internal clocks of a CPU core is disabled, thus the dynamic power consumption is reduced. In C3 state, the power consumption is further reduced by cutting the clocks of entire CPU core, which reduces the dynamic power to 0. However, to transit back to the normal operation mode, it will spend 59 μsec, compared to only 1 μsec for C1-C0 transition. While in state C6, the power supply to the core is gated to reduce both dynamic and static power consumption to approximately zero. The wakeup time a core in idle state C6 is the longest at 80 μsec. Table 4.1 shows the power consumption of each C-state. We can see that the deeper C-states consume much lower power than the active state (C0). Also, the transition overhead of entering that C state increases, from 1 to 80 μsec, as the C-state gets deeper, since it takes more time to disable/enable the corresponding components. Moreover, the power consumptions of the C-states, except for C6, are temperature dependent because of the static power consumption.

Considering busy/idle execution pattern and different C-states, the power consumption of the multicore processor can be formulated as:

$$P_{processor} = \sum_{n=1}^{N} \frac{P_a^n \times t_a^n + \sum_{i \in C} \left(P_i \times \left(t_i^n - t_{tran,i}^n\right) + P_{tran} \times t_{tran,i}^n\right)}{t_a^n + \sum_{i \in C} t_i^n} \tag{4.4}$$

, where the super-script n is the index to the core, and N is the number of cores in the processor. The first term in the denominator is the active energy consumption, P_a is the active power consumption; P_i is the idle power consumption in ith C-state and P_{tran} is the maximum power spent during the state transition, which makes the second term the idle energy consumption. t_a, t_i and $t_{tran,i}$ are the length of active period, the time spend in ith C-state and the state transition time, respectively.

4.2.4 Problem with Existing Idle state Linux Governor

The goal of the idle power management is to minimize the second term in equation (4.4). Because of the state transition overhead, a CPU core should enter the C-states if the idle duration is greater than a threshold, called energy break-even time (EBT), to avoid inefficient usage of C-states, and this threshold increases as the C-state gets deeper, as shown in Table 4.1. As a result, the idle period should be known beforehand to select the most energy efficient C-state a core should enter. In the Linux kernel, the prediction-based sleep-state selection algorithm, called menu idle governor [3], is used for this purpose. It uses a history-based idle length predictor. If the variation among the past idle intervals is small, the average of past idle intervals will be used as the prediction; otherwise the next scheduled OS timer event will be used as the prediction. However, due to the *unpredictable nature of the request arrival*, the menu governor or any history based idle period predictor is not very useful with the non-deterministic execution pattern [38], where over-prediction (predicted idle period is shorter than the actual one) will cause energy inefficiencies and under-prediction will cause unnecessary latency increase and power wastage due to the large state transition overhead.

Besides the difficulties for the idle period prediction, another issue for idle power management lies in the *frequent and random* ON/OFF pattern of network and datacenter applications, which greatly affect the effectiveness of using C-states. Fig. 4.3 gives the experimental results of the power consumption of a network application under different traffic loads. There are three scenarios in Fig. 4.3: *"Linux"* represents the power consumption of using Linux prediction-based C-state selection algorithms (menu idle governor), and the

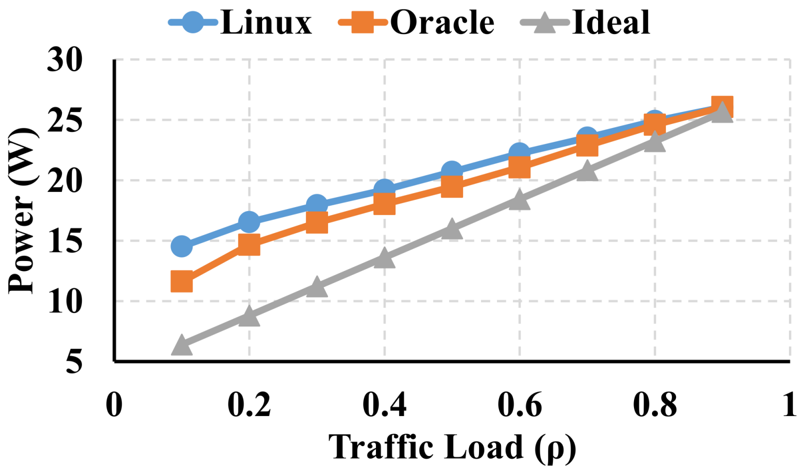
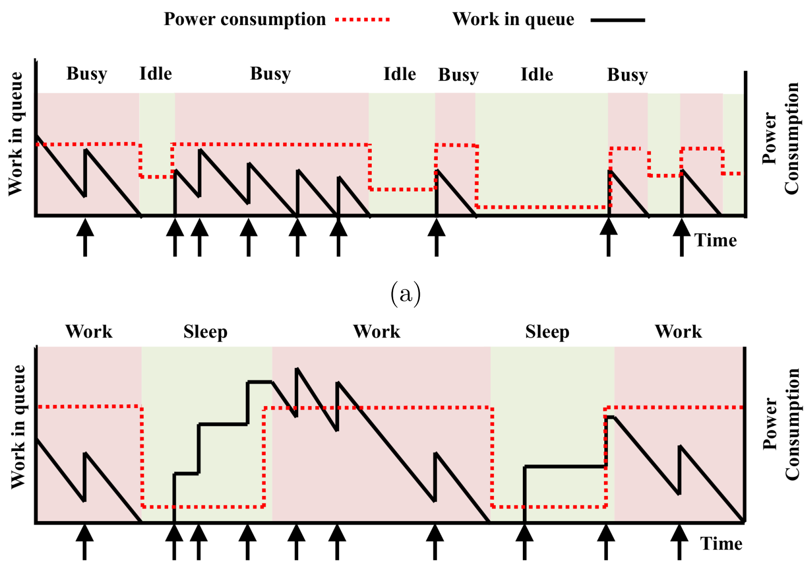


Figure 4.3: Power consumption under different scenarios.

results are measured using on-chip energy sensors; *"Oracle"* represents the case when all idle durations are known beforehand so the core can enter the most energy efficient C-state with no prediction error, and *"Ideal"* gives the power consumption with perfect energy proportionality, where all CPU cores go into deepest sleep state during all of its idle time with zero transition overhead. One should notice that the Ideal scheme serves as a theoretical lower bound of the power consumption that can be achieved. In Fig. 4.3, we can see that when the system has highly activity ($\rho > 0.7$), the difference between Linux and Oracle cases are small, because the idle period is mostly short, which make shallowest C-state the most energy efficient. However, due to the inaccurate prediction, the Linux prediction algorithm becomes less effective when the traffic activity decreases, 16% and 25% more power consumption than the Oracle case at = 0.4 and 0.1, respectively.

Furthermore, despite the perfect idle state selection of the Oracle case, the existence of short idle periods will prevent the system from going into deep C-state. For this reason, even with low activity, the system operates far from being energy proportional i.e.,



(a)

(b)

Figure 4.4: The packet/request processing and power consumption pattern under (a) default and (b) smart sleep schemes.

consumes power proportionally to its utilization, as shown in the *Ideal* case.

4.3 Smart Sleep

In the original system, CPU cores are busy processing requests in the queue. They become idle when their queues are empty and spend most of their time idle when the traffic is not heavy. The state-of-the-practice idle power managing algorithm is unable to exploit the power saving opportunities during short idle periods due to the relatively high overhead of the deep idle state transition, as shown in Fig. 4.4(a). In this section, we present a technique that can efficiently utilize processor deep sleep by smartly selecting the duration of its sleep. This section describes our power saving scheme, derives per-core analysis, and studies its impact on the other performance metrics. The results will be applied to multiple cores in the next section.

We propose Smart Sleep, an approach to reshape the request processing pattern by creating longer sleep periods and allowing the CPU cores to enter deeper idle states that consumes less power. There are two states for a CPU core in our scheme: working and sleeping. In working state, the CPU core fetches requests from the queue and processes them until there is no request left in the queue. When the queue is empty, the CPU core will switch to sleeping state and buffers all the incoming requests in the queue. After a predetermined sleep time, the core wakes up and goes back to the working state, and continues to serve requests in the queue.

Smart sleep solves two major issues with the existing idle power management scheme: short and unpredictable idle period. As shown in Fig. 4.4(b), in the smart sleep scheme, the incoming traffic, from the cores perspective is reshaped into bursts of arrival. CPU core wakes up to process a batch of requests, and then sleeps for a longer time. The intent is to provide sufficient idle time for cores, so that they can enter the deeper C-state efficiently, and improve idle period power saving. Also, since the sleep period is determined before the idle, the exact length of the idle period is known, and the idle state selection become straightforward. However, accurate determination (prediction) of the sleep period is critical to the efficiency of smart sleep technique. We will describe the prediction method in the next section. In the rest of this section, we focus on two major performance metrics: power consumption and response time, and theoretically analyze them under smart sleep scheme. Experimental verification of our analytical techniques is given in section 4.5.

4.3.1 Power Consumption

To understand the effectiveness of the idle power saving with smart sleep, we first derive the idle power consumption during the sleep period. Let t_{sleep} be the sleep time, P_c be the power consumption during the core idle period and is dependent on the C-state, t_{tran} and P_{tran} be the time and power spent during the state transition. Since the sleep period is determined before the CPU goes to idle, the best suitable C-state is known beforehand, and the average power consumption during sleep period, P_{sleep}, can be formulated as:

$$P_{sleep} = \frac{P_c \times t_{sleep} + (P_{tran} - P_c) \times t_{tran}}{t_{sleep}} \tag{4.5}$$

Since our smart sleep just rearranges the processing pattern and does not change the overall busy or idle time, the overall power consumption can be formulated as:

$$\begin{aligned} P_{avg} &= P_{work} \times \rho + P_{sleep} \times (1 - \rho) \\ &= P_{work} \times \rho + P_c \times (1 - \rho) \\ &+ (P_{tran} - P_c) \times \frac{t_{tran}}{t_{sleep}} \times (1 - \rho) \end{aligned} \tag{4.6}$$

4.3.2 Response time

The smart sleep introduces a sleep period whenever the core becomes idle, and request processing is deferred to the end of this period. Thus, the response time of the requests will increase. From the vacation queue theory [113], the latency, L_{sleep} can be decomposed into the sum of two independent random variables,

$$L_{sleep} = L + L_d \tag{4.7}$$

, where L is the latency of a classical M/G/1 queue without sleep, and L_d is the additional delay due to the introduced sleep period, and is the residual time of a sleep period.

Average response time: From equation (4.7-4.10), the expected response time, $E(L_{sleep})$, can be formulated as:

$$E(L_{sleep}) = \frac{\lambda E(S^2)}{2(1-\rho)} + E(S) + \frac{t_{sleep}}{2} \tag{4.8}$$

The first term of equation (4.8) is the Pollaczek-Khinchin formula for the expected queuing delay in a standard M/G/1 queue, the second term, $E(S)$, is the expected service time, and the final term is the additional delay caused by the sleep, and is proportional to t_{sleep}.

Response time distribution analysis: Besides obtaining the closed form equation for the average latency, we also need to model the latency distribution. The QoS of datacenter applications is usually defined by the tail latency (i%ile latency), where i% of requests must be finished before a given time threshold. To acquire the tail latency, we need to further model the distribution of the L_{sleep}. While obtaining the closed form equation of the latency distribution for the M/G/1 queue is analytically intractable, we provide some theoretical analyses which give insight on building the latency behavior model under smart sleep scheme,

Let S be the random variable of the service time and $f_L(x)$ and $f_{L_d}(x)$ be the probability density function (PDF) of random variable L and L_d, respectively. From equation

(4.7), we will have

$$\begin{aligned} f_{L_{sleep}}(x) &= f_L(x) * f_{L_d}(x) \\ &= \int_{-\infty}^{\infty} f_L(x) f_{L_d}(z-x)dz \end{aligned} \tag{4.9}$$

Thus if we can acquire the PDF of L and L_d, we can compute the latency distribution of $f_{L_{sleep}}$ and the tail latency.

First, the latency distribution of a classical M/G/1 queue, L, can be modeled as:

$$L^*(s) = W^*(s)S^*(S) = \frac{(1-\rho)s}{s-\lambda+\lambda S^*(s)} S^*(s) \tag{4.10}$$

,where W is the waiting time, $L^*(s)$, $W^*(s)$ and $S^*(s)$ is the laplace transform of L, W and S. Also, by rearranging $W^*(s)$, the random variable W can be decomposed into:

$$W = \frac{(1-\rho)}{1-\rho R} = \sum_{n=0}^{\infty} \rho^n R^n \tag{4.11}$$

, where R is the residual service time in the server conditioned on that there is a customer in the server, and R^n is the n-fold convolution of R. Furthermore,

$$R^*(s) = \frac{1-S^*(s)}{sE(S)} \Rightarrow f_R(x) = \frac{1-F_S(x)}{E(S)} \tag{4.12}$$

From equation 4.11 and 4.12, we can derive $f_L(x)$ with only the service time distribution.

Now, we need to have the distribution of L_d. Since L_d is the residual time of a constant sleep period t_{sleep}, $f(L_d)$ will be a uniform distribution

$$f_{L_d}(x) = \begin{cases} \frac{1}{t_{sleep}}, & \text{if } 0 \leq x \leq t_{sleep} \\ \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

Response time distribution model: From equation (4.9) and (4.13), we can see that $f_{L_{sleep}}(x)$ is the *sliding window summation* of the distribution $f_L(x)$, with the window width equal to t_{sleep}. Figure 4.5 gives an example of how to form $f_{L_{sleep}}(x)$. First, let ϕ be

$$\phi = \min\{k : f_L(x) < \epsilon \quad \forall x \geq k\} \tag{4.14}$$

By setting ϵ to be small (<0.0001), ϕ helps us to simplify the analysis without sacrifice the accuracy.

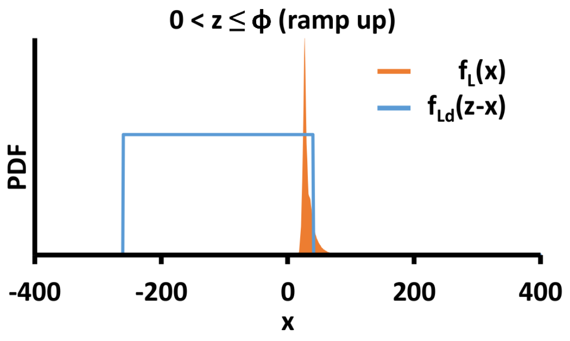
The convolution in equation (4.9) can be broken down into five phases based on z.

$\mathbf{z < 0}$: In this phase, there is no overlap between f_L and f_{L_d}, thus $f_{L_{sleep}}(z) = 0$.

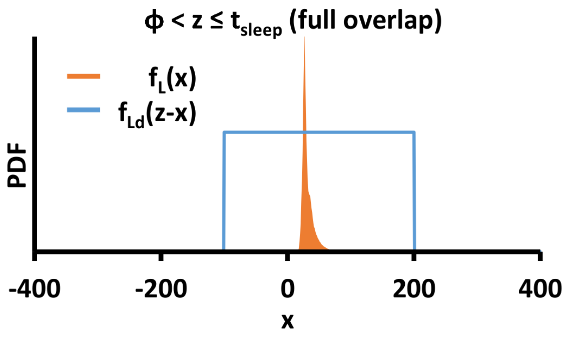
$\mathbf{0 < z \leq \boldsymbol{\phi}}$: As shown in 4.5a, f_{L_d} is moving toward f_L, resulting partially overlapped. As f_L shifting right (increasing z), the overlapped region increases, thus we can have the following formulation:

$$\begin{aligned} f_{L_{sleep}}(z) &= \frac{1}{t_{sleep}} \int_{-\infty}^{z} f_L(x)dx \\ &= \frac{1}{t_{sleep}} F_L(z) \end{aligned} \tag{4.15}$$

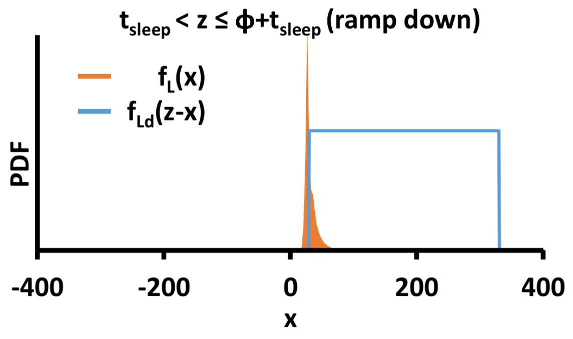
. The PDF of the $f_{L_{sleep}}$ is equal to $F_L(x)$, the cumulative distribution function (CDF) of



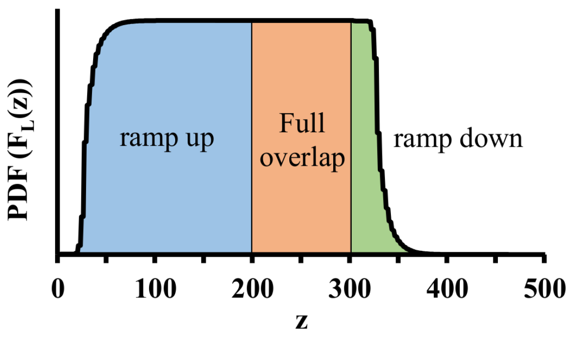
(a) Ramp up phase



(b) Full overlap phase



(c) Ramp down phase



(d) the distribution $f_{L_{sleep}}(z)$

Figure 4.5: Illustration of $f_{L_{sleep}}(z)$ through sliding window summation of $f_L(x)$

L.

$\boldsymbol{\phi < z \leq t_{sleep}}$: In this range, whole f_L is overlapped with the f_{L_d}, thus $f_{L_{sleep}}(z) = \frac{1}{t_{sleep}}$.

$\boldsymbol{t_{sleep} < z \leq \phi + t_{sleep}}$: In this phase, f_{L_d} is moving away from the f_L, resulting in partial overlap. Contrary to the ramp up phase, the overlapped region decreases as f_L shifting right (increasing z). $f_{L_{sleep}}(z)$ can then be formulated as:

$$\begin{aligned} f_{L_{sleep}}(z) &= \frac{1}{t_{sleep}} \int_{z-t_{sleep}-\phi}^{z-t_{sleep}} f_L(x)dx \\ &= \frac{1}{t_{sleep}}(1 - F_L(z - t_{sleep})) \\ &= \frac{1}{t_{sleep}} \overline{F}_L(z - t_{sleep}) \end{aligned} \tag{4.16}$$

. The PDF of the $f_{L_{sleep}}$ is equal to $\overline{F}_L(x)$, the complementary cumulative distribution function (CCDF) of L, as shown in 4.5c.

$\boldsymbol{\phi + t_{sleep} < \mathbf{z}}$: In this region, since $F_L(x)$ is small (with the definition of ϕ) , thus $F_{t_{sleep}} \approx 0$.

In summary, we have the model of PDF of L_{sleep} as

$$\begin{aligned} &f_{L_{sleep}}(z) = \\ &\frac{1}{t_{sleep}} \times \begin{cases} F_L(z), & 0 \le z \le \phi \\ 1, & \phi < z \le t_{sleep} \\ \overline{F}_L(z - t_{sleep}), & t_{sleep} < z \le \phi + t_{sleep} \\ 0, & otherwise \end{cases} \end{aligned} \tag{4.17}$$

Note that the above analysis assumes that $\phi > z$. This assumption may not be valid for high traffic load and/or small t_{sleep}. As shown in fig 4.6, the ramp up and ramp down phases are now limited by $0 \le z < t_{sleep}$ and $t_{sleep} \le z < \phi + t_{sleep}$. In between, there is an *sliding overlap* phase, where only partial F_L is summed. While computing the sliding overlap phase may be time consuming, we develop an approximation based on the following

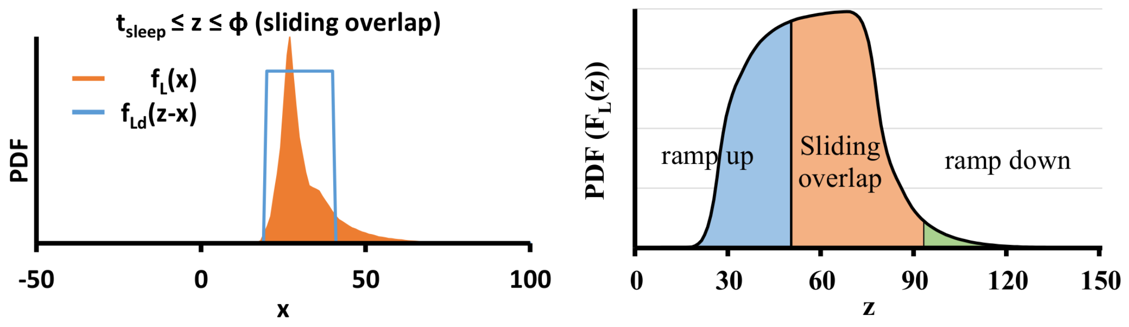


Figure 4.6: Illustration of sliding overlap when $\phi \geq z$.

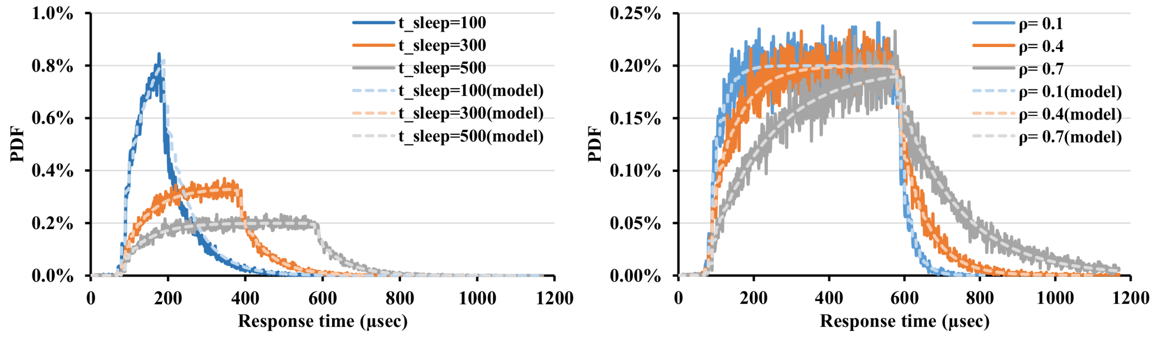
observations:

$$\begin{aligned} f_{L_{sleep}}(z) &\leq \frac{1}{t_{sleep}} F_L(z) \\ f_{L_{sleep}}(z) &\leq \frac{1}{t_{sleep}} \overline{F}_L(z - t_{sleep}) \end{aligned} \tag{4.18}$$

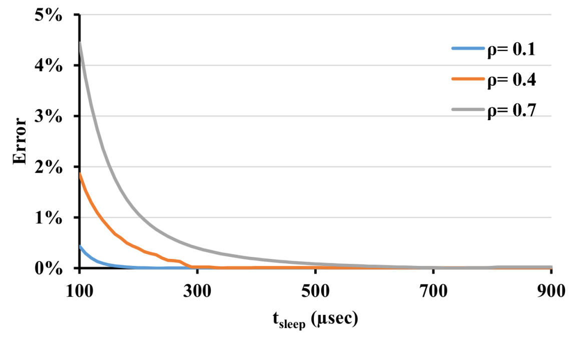
, and the approximated distribution, $f'_{L_{sleep}}(z)$ is

$$f'_{L_{sleep}}(z) = \frac{1}{t_{sleep}} \min(F_L(z), \overline{F}_L(z - t_{sleep})) \tag{4.19}$$

Figure 4.7 gives the results of our response time distribution model. We also present the actual experimental results to show the accuracy of our model. From fig 4.7a, we can see that our model can faithfully represent the distribution at high t_{sleep}. At low t_{sleep}, our approximated distribution model slightly overestimated the distribution. Similarly, in fig 4.7b, our approximation model overestimates at high load. Figure 4.7c gives the error between the tail (95^{th}) latency derived from our model and the experimental results. The error is defined as $(tail_{model} - tail_{experiment})/tail_{experiment}$. We can see that our model overestimates the tail latency under smart sleep scheme, especially at high load and low



(a) (b)



(c)

Figure 4.7: Experimental and model results for response time distribution under smart sleep. (a) different sleep periods at fixed traffic load $\rho = 0.4$, (b) different traffic load with the same sleep period $t_{sleep} = 500\mu$sec, and (c) the modeling error for different traffic load and sleep periods.

t_{sleep} because of the nature of our approximation. One thing to notice is that the minimum t_{sleep} we choose is larger than 156 μsec (the EBT of C3 state). As a result, the worst case modeling error will be less than 3%.

4.4 Multicore Smart Sleep Framework

In section 4.3, we presented the detailed performance analyses of our smart sleep in the CPU core level. However, to apply it to the multicore processor, one more dimension needs to be considered, ie. the number of cores that need to be active. As the traffic increases, more cores need to be active to maintain a certain level of the response time. However, turning on the minimum number of cores based on the core utilization and response time might not give the best power saving. Although turning off a core will save the state transition power (staying at C6 state without waking up), it will shorten the sleep periods for the active cores and increase their power consumption. In this section, we will analyze this problem and answer two major questions: (1) When does a core need to be turned on/off, and (2) how is the sleep time of the active cores determined so that the maximum power saving will be achieved under a given response time constraint. First, we will theoretically analyze the performance of multicore smart sleep scheme, and then propose a runtime algorithm to dynamically determine the number of active cores in the processor and sleep time for each core based on prediction of the future traffic. Experimental verification of our analytical techniques is given in section 4.5.

4.4.1 Core Sizing

In this section, we theoretically determine the optimum configuration of the active cores of the multicore processor and the per-core sleep time. The optimal configuration is the one that minimizes power consumption while meeting a target QoS (response time) constraint. The server should be running full out, i.e., activating all cores, under default scheme to meet the QoS constraint during periods of peak demand. In contrast, at lower loads there is slack in meeting the QoS which can be exploited to reduce power as much as possible. We set the QoS constraint as the response time under the peak load ($\rho_c = \lambda_c/\mu$) without applying any power management techniques, where CPU will not enter any power saving state during idle. With the number of cores N, the response time constraint, L_c, will follow the standard M/G/1 queuing model with load ρ_c/N, assuming the load is evenly distributed to each PE. Also, equation 4.6 shows that longer sleep period will give lower power consumption, our optimization can then be transformed as:

$$\begin{aligned} &\text{maximize} \quad t_{sleep} \\ &s.t., \quad L(\rho/k, k, t_{sleep}) \leq L_c, \text{ and } 1 \leq k \leq N. \end{aligned} \tag{4.20}$$

, where P^i is the power consumption of core i, k is the number of active cores, and ρ/k is the load/utilization of each active core. We propose a power optimization heuristic in the following section.

4.4.2 Power Optimization Heuristic

Constraint on average latency: Since the average latency under smart scheme has a closed form solution, we develop an heuristic to find the best configuration to minimize power under average latency constraint. Since the core consumes no power while being inactive (put into C6 state), and all active cores are distributed with same traffic load and sleep time, the total power consumption is equal to k times of a core under the sleep time t_{sleep} with load $\lambda/k\mu$. By combining equation (4.6) and (4.8), the maximum sleep time with k active cores is:

$$t^k_{sleep} = E(S^2) \times \left[\frac{k\lambda_c - N\lambda}{(N-\rho_c)(k-\rho)}\right] \tag{4.21}$$

Now, we need to determine the number of active cores to achieve minimum power consumption. In general, we want to turn off as many cores as possible to save the power consumption. However, the more cores we turn on, the longer the sleep time each core can have, and the lower the power consumption of each core. There will be some cases that having more active cores, with longer per-core sleep time, will consume less power than having less active cores with short sleep. These two competing factors must be analyzed and balanced.

Let the sleep time with n and k active core be t^n_{sleep} and t^k_{sleep}, where $k > n$. Because of the transition power overhead, we know that the power consumption of n active cores will be less than that of k cores if both t^n_{sleep} and t^k_{sleep} will drive the cores into the same C-state. On the other hand, if t^k_{sleep} is long enough to drive CPU cores into deeper C-state than t^n_{sleep}, the choice becomes complicated, depending on the load, C-states and the value of n. We propose a simple heuristic to find the configuration that minimizes the power consumption as follows:

Let $k^i = \{k^i_1, k^i_2, ..., k^i_n\}$ be the set of configuration (number of core) where $k^i_n > k^i_{n-1} > ... > k^i_1$ and $EBT_{i+1} > t^{k^i_n}_{sleep} > EBT_i$ so they will enter the same C-state i. We know that the minimum power consumption in the set K^i is the configuration which has minimum number of active core (k^i_{min}). As a result, to find the configuration with minimum power consumption for all possible configuration, we just need to check one configuration in each set k^i for each C-state i. By replacing t_{sleep} with EBT_i, the energy breakeven time for C-state i, we can compute the minimum power configuration of the set k^i, , as:

$$k^i_{min} = \left\lceil \rho \frac{EBT_i(N - \rho_c) - \frac{E(S^2)}{E(S)} N}{EBT_i(N - \rho_c) - \frac{E(S^2)}{E(S)} \rho_c} \right\rceil \tag{4.22}$$

Once we know k^i_{min}, we can compute their corresponding sleep period and power consumption through equation (4.21) and (4.6), and find the configuration which minimize the power

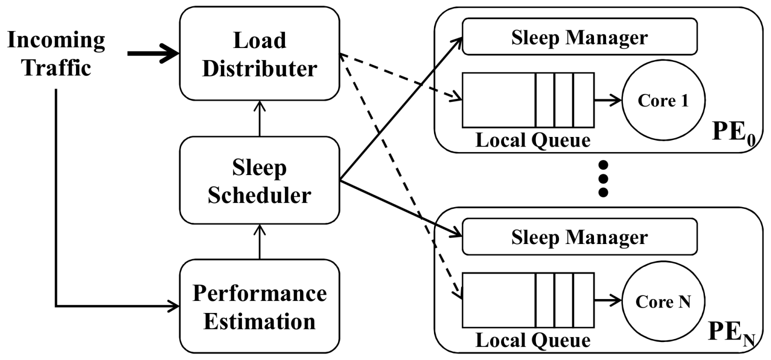


Figure 4.8: The overview of multicore smart sleep framework.

consumption.

Constraint on tail latency: Our power optimization heuristic can also adapt to different latency constraints, such as tail latency, with some modifications. Instead of having closed form equation (equation (4.22)) to compute the configuration under average latency, we need to use the PDF model (equation (4.17) and (4.19)) we built in section 4.3.2 to find k^i_{min} for each Cstate i.

4.4.3 Runtime Framework

In this subsection, we propose a power aware scheduling runtime, which monitors and predicts the incoming traffic, and dynamically chooses a configuration. It determines the number of cores that needs to be active and the sleep time of each core. With the selected configuration, we can achieve maximum power saving while satisfying the QoS requirement.

Fig. 4.8 shows the system overview of our system framework, which is composed of a scheduler and many processing engines (PEs). A PE essentially consists of a processing core and its input buffer (queue), as shown in Fig. 4.8. The arriving packets will be distributed to the designated PE as assigned by the scheduler. The core in the PE will fetch and process the requests from its local queue following the smart sleep scheme. The aims of the scheduler are: (1) predict the future traffic load, (2) choose the optimum configuration tuple to minimize the power consumption without violating the QoS constraint, and (3) distribute workload to the PEs. Since the future arrival rate is unavailable, we use the history of traffic pattern to predict the future traffic load. We denote this predicted arrival rate as $\overline{I}$, and estimate it with the measured history of past arrivals, denoted as I. We employ an Exponentially Weighted Moving Average (EWMA) filter:

$$\overline{I_k} = \alpha\overline{I_{k-1}} + (1-\alpha)I_k \tag{4.23}$$

EWMA is a very low-overhead filter that places more importance on recent data and discounts older data in an exponential manner. The value of the parameter α dictates the degree of filtering, a constant smoothing factor between 0 and 1. A higher α discounts older observations faster. In our experiment, we find out that setting α to 0.1 and window size to 30 seconds provides a good balance between filter agility and stability. Although EWMA is a good traffic predictor most of the time, a more sophisticated predictor might improve the accuracy, which we plan to explore in our future work.

The predicted traffic will then be used for choosing the optimum configuration. First, it finds the minimum number of cores needed to be active ($\lceil \rho/\rho_c \rceil$). Then, based on equation (4.22), it computes and compares the minimum power consumption under each Cstate, and finds out the optimum number of active cores and per-core sleep time. Once the configuration is determined, it will be sent to the sleep managers of each PE. The per-core manager will apply the smart sleep and regulate the request processing of the core. Finally, the incoming requests are distributed among the active cores in round-robin fashion to the active cores, as done by Linux scheduler. Although more sophisticated scheduling algorithms can be applied considering packet reordering, it is out of the scope of this paper.

4.5 Performance Evaluation

In this section, we first give our experiment setup in subsection 4.5.1, and briefly give some results for the single core in subsection 4.5.2. Then we present the evaluation of multicore smart sleep and our runtime framework in subsection 4.5.3 and 4.5.4.

4.5.1 Experiment setup

Our experimental system consists of a traffic generator and a processing server with an Intel Quad-core processor connected through 10 Gbps Ethernet link. The CPU cores are kept at a fixed frequency, 3.4GHz (the maximum frequency of the processor), to eliminate run-to-run performance variance. Our Quad-core processor is equipped with on-chip energy sensor which records the energy consumption of all four cores, and the readings are collected from accessing machine specific register (MSR) at the rate of 1 kHz.

We consider both synthetic traces and real traces of network/data center applications to evaluate our proposed schemes. The synthetic traces are considered in order to observe the impact of traffic load on power consumption and latency, which cannot be done using real traces. We generate synthetic traces of 10 million packets/requests, assuming the arrival pattern is Poisson, and service time is exponential distribution. The traces are re-played by the traffic generator, sent over the Ethernet to the server, and the energy consumption, total execution time and per-request response time of the processing server are recorded. The range of the sleep time is selected to be from 100 to 900 μsec. We do not increase the sleep time further because increasing the sleep time beyond 900 μsec only marginally improves the power saving. The core is turned off by removing cores from active working set based on kernels built-in CPU “hotplug” support. Thus OS will not schedule any task to the off cores.

Besides using synthetic trace, we also use some real-world traces to evaluate our proposed framework. We are interested in evaluating the performance of our system with a broad range of applications. To this end, we run experiments with the service rates obtained from executing four network and datacenter applications: *IPv4 routing, URL, IPSEC and memcached*. IPv4 routing and IPSEC are commonly deployed network packet processing applications, URL is an application aware switching algorithm applied in the server farms, and memcached is a popular key-value store datacenter application. One thing to notice is that the QoS of memcached is defined by the tail latency to ensure the performance stability, where the latency will be lower than the constraint most of the time [47]. We choose 95%ile latency as our tail latency, this different QoS target can be easily adapted in our smart sleep scheme by changing the response time model and the derivation algebraically. We executed the benchmark applications on our server and collected the statistics about service time and power consumption on our server, as given in Table 4.2.

For datacenter applications, we use the trace provided in the Cloudsuite [47];

Table 4.2: Characteristic of packet processing and datacenter applications.

Name	Functionality	E(s)
Memcached	In-memory key-value store	20μsec
IPv4 Routing	Radix based IPv4 routing	50μsec
URL	Content-based switching	105μsec
IPSEC	Packet encryption using AES-128	152μsec

for network applications, we use the packet traces of real network from CAIDAs equinix monitors [2] as inputs. CAIDA captures the traffic traces on OC192 links. The traces in this data set are of 60 minutes long duration. They were captured on June 19th, 2014, containing the time stamp and header of packets. CAIDA also makes real time traffic load available online, and we record one day data as the traffic trace. These traces are replayed and scaled with actual traffic load in our traffic generator to model the real traffic scenario.

4.5.2 Single Core Results

Power consumption: Fig. 4.9 shows experimental results of the power consumption under smart sleep and the oracle Linux power management scheme as a function of the traffic intensity. In these experiments, we run the application *URL*, with the average service time 105 μsec and vary the average inter-arrival times according to the traffic load we need. The requests inter-arrival time follows the exponential distribution. Similar to Fig. 4.3, we plot the power consumption under four different schemes: the default Linux power management scheme, labeled as "Linux", the oracle C-state selection scheme, "Oracle", the smart sleep scheme and the energy proportional power consumption, labeled as "Ideal". For the smart sleep scheme, we give two sets of results with $t_{sleep} = 300$ and 900 μsec, labeled as Sleep - 300μs and Sleep - 900μs, respectively. For the Oracle scheme, we used pyTimechart [8] to collect system busy/idle traces. The total energy consumption

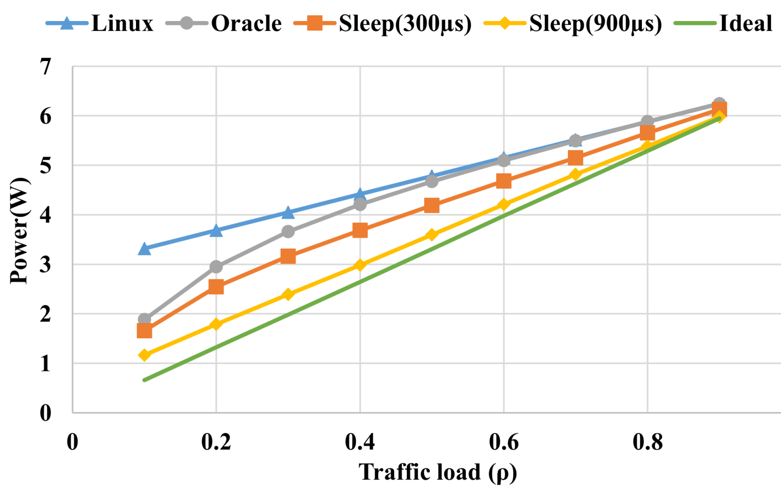


Figure 4.9: Power consumption under different scenarios.

under Oracle scheme is then the weighted sum of the busy periods and each idle periods with the most energy efficient sleep states (given in Table 4.1).

First, we can see that significant power consumption gap exist between C-states selection schemes (Linux and Oracle) and the energy proportion scheme (Ideal). The Linux power management does not provide good energy performance mainly because of the idle period prediction error and the CPU core will hardly ever go to C-states deeper than C1, resulting poor energy efficiency. However, even with most energy efficient C-state selections, as in the Oracle scheme, the power-load curve is still super-linear and far from being energy proportional. The reason is that the core can only enter deep sleep state and save power if and only if the idle period is longer than the EBT, and the probability of that increases non-linearly as the load decreases. In contrast, the power-load curve for the smart sleep is linear with a fixed t_{sleep}. Second, the smart sleep scheme can achieve significant power saving even compared with the oracle C-state selection scheme. It is because the forced idle sleep periods will drive the core into C6 state with 0 idle power. When $t_{sleep} = 900$ μsec,

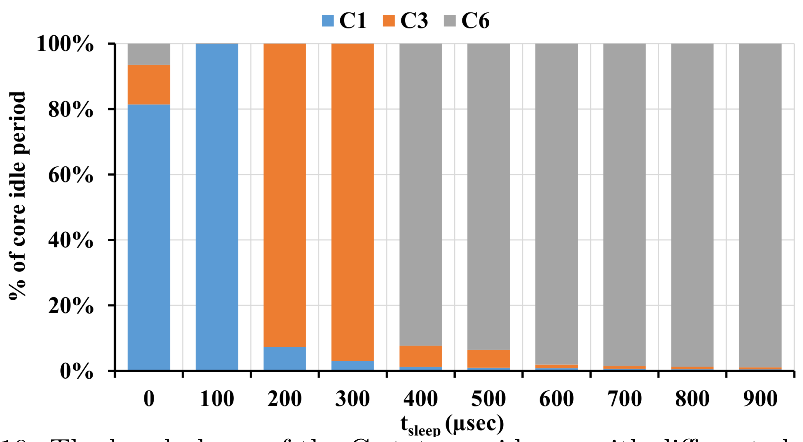


Figure 4.10: The break down of the C-states residency with different sleep period.

we can achieve an average 23% and maximum 41% power saving. Furthermore, with short sleep period (300 μsec), which barely satisfies the C6 state threshold, we can still achieve 10% power saving over the oracle scheme. The power saving of smart sleep scheme is even more significant compared with the state-of-the-practice Linux scheme, 21% and 35% power saving on an average with 300 and 900 μsec sleep period, respectively.

C-state residencies: To show that the core will enter deeper sleep state during the sleep period, we also record the time spent in each C-state during the experiments. Fig. 4.10 gives the percentage of the idle time that the CPU core spent in each C-state at medium traffic load ρ=0.4, with $t_{sleep} = 0$ representing the default Linux scheme, where it puts CPU cores to C3 and C6 states only for small portion of time, less than 20%, because of the nature of short idle periods and the inaccurate idle period prediction. On the other hand, the smart sleep scheme will introduce a predictable idle period pattern, which allows the underlying idle governor to enter the most energy efficient idle state without any prediction error. For smart sleep scheme with $t_{sleep} = 100$ μsec, the core will spend all its idle time in C1 state because of the short sleep time. The core will wake up and check the queue every 100 μsec,

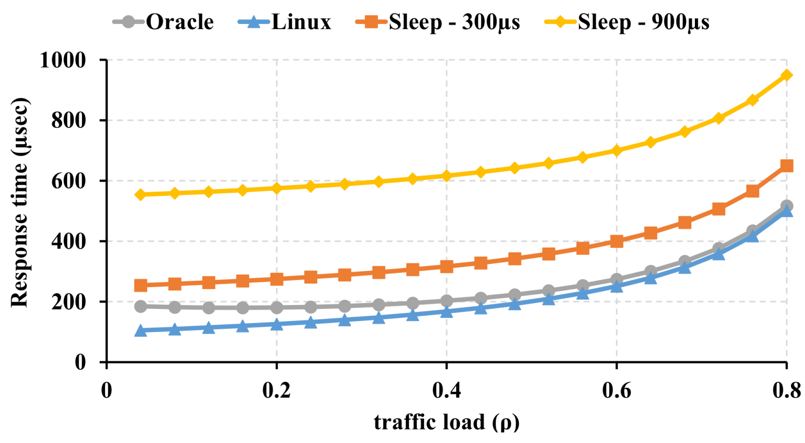


Figure 4.11: Average response time under different scenarios.

which prevents the core from entering C3 and C6 states completely. As t_{sleep} increases to 200 and 400 μsec, the core will enter C3 and C6 during its idle periods, respectively. These results show that our smart sleep scheme can effectively save idle power of a CPU core by utilizing deep C-states.

Response time: The power saving in the smart sleep scheme does not come free. It increases the response time. The idea is that as long as the latency does not exceed the limit, it is acceptable to delay the response and save power. Fig. 4.11 gives the average response time under different power management schemes with traffic load increasing from 0.1 to 0.9. In the Linux scheme, most of the time, the core will go to C1 state during idle period with negligible transition time, hence the response time is minimum. In the Oracle scheme, the core enters the deep C-states while the traffic is low, the latency is slightly increased because of the C-state wakeup transition time. However, as the traffic increases, the frequency of entering deep C-states decreases, and the response time of the Oracle scheme is close to the Linux scheme. As expected, additional response time is introduced by the smart sleep, and is around half of the sleep time compared with Linux scheme. The

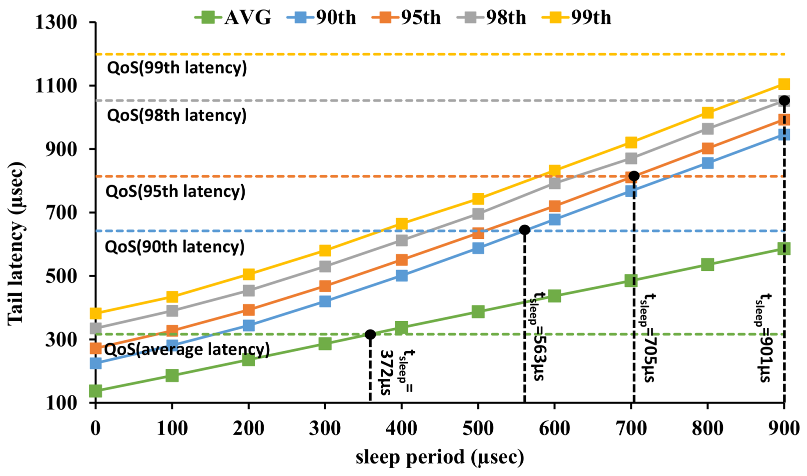


Figure 4.12: Different tail latencies under smart sleep scheme.

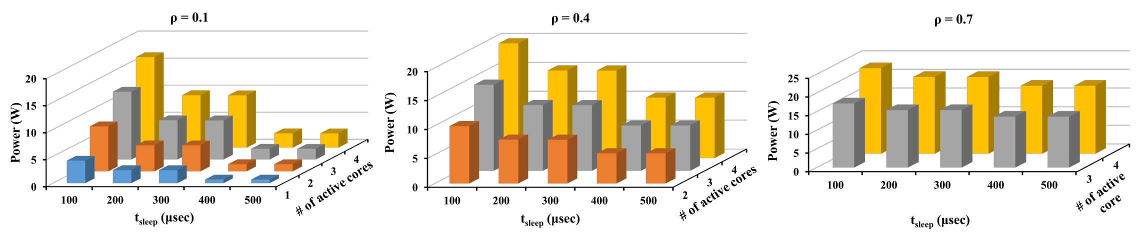
sleep time can be varied depending on the allowable response time for the packets. Note that the tail response time (higher percentile response time) will follow the similar trend as in Fig. 4.11.

Target tail latency: As we mentioned previously, some network and datacenter applications will define their service quality with high percentile latency (tail latency). Fig. 4.12 gives the experimental results of tail latency under smart sleep. In this figure, we show the average latency along with four different levels of tail latency: 90th, 95th ,98th and 99th percentile latency. The solid lines represent the tail latency with smart sleep scheme under load 0.4. The horizontal dotted lines represent the QoS constraints under each tail level. The QoS constraints defined as the average/tail latency under default scheme with load 0.8. The intersection of dotted and solid lines (black dots) for each tail latency level represents the corresponding sleep period length that we should choose in smart sleep scheme. We can see that with higher percentile QoS constraint, longer sleep period can be used in smart sleep scheme. This is because the QoS with high percentile latency is much larger than that with average latency. Compared with 316 μsec target average latency, the target tail latency increases from 640 μsec (90th) to 1200 μsec (99th). Also, the tail latency overhead introduced by smart sleep scheme increases at similar rate with different levels of tail latency. As a result, the sleep period increases significantly from 372 to more than 900 μsec as tail percentile increases. Also, combining with the fact that longer sleep period will lead to lower power consumption (Fig. 4.9), we can conclude that smart sleep scheme can achieve higher power saving with the higher percentile tail latency constraint.

4.5.3 Multicore Results

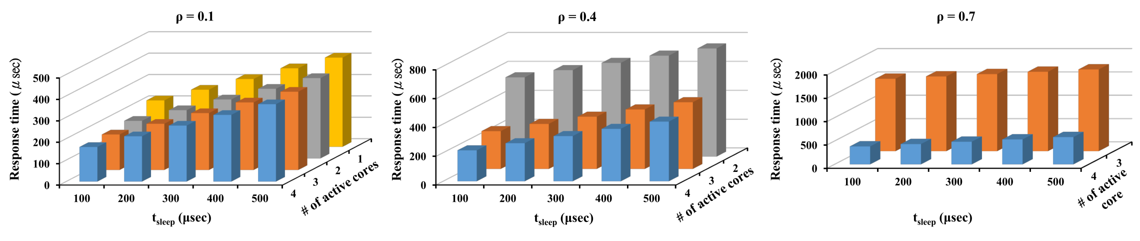
When there are multiple cores in CPU, the number of active cores can be varied in addition to the sleep times in a smart sleep scheme. In this section, we first give the experimental results for different configurations of multicore smart sleep scheme under different traffic loads. Then we show the effectiveness of our runtime with real world traces. We use the notation (V, k) to represent the configuration of $t_{sleep} = V\mu$sec and k active cores.

Fig. 4.13 and 4.14 give the experimental results of power consumption and response time in our multicore smart sleep scheme, running URL application with synthetic traffic traces. In these figures, the x-axis is the length of the sleep time, y-axis (depth) is the number of active cores, and z-axis (elevation) is the power consumption or response time. We only give the results with $t_{sleep} \leq 500\mu$sec because the results change marginally with higher sleep times. In Fig. 4.13, we can see two noticeable drops of the power consumption (at $t_{sleep} = 200$ and 400 μsec) as sleep period increases. It is because the CPU core should enter a specific C-state only when the idle time is longer than the EBT of that C-state. Also, the minimum number of active cores is increasing as the load is increased, simply



(a) (b) (c)

Figure 4.13: Power consumption after applying multicore smart sleep scheme with different configuration under (a) ρ=0.1, (b) ρ=0.4 and (c) ρ=0.7 traffic load.



(a) (b) (c)

Figure 4.14: Response time after applying multicore smart sleep scheme with different configuration under (a) ρ=0.1, (b) ρ=0.4 nad (c) ρ=0.7 traffic load.

because it needs more processing capability.

An important observation is that different configurations may yield to similar power consumption. For example, (200, 1) and (500, 4) in $\rho = 0.1$, and (300, 2) and (500, 3) in $\rho = 0.4$ give similar power consumption results, but the response time of these configurations are very different. As shown in Fig. 4.14, the response time is increasing as the sleep period is increased. Also, as the number of active cores increases, the load on each core is decreased. It is clear that there exists a tradeoff between power consumption and response time in choosing the number of active cores. Small number of active cores will give better power consumption but worse response time compared with the large one. Thus, finding the best configuration is crucial in the multicore smart sleep. Our smart sleep

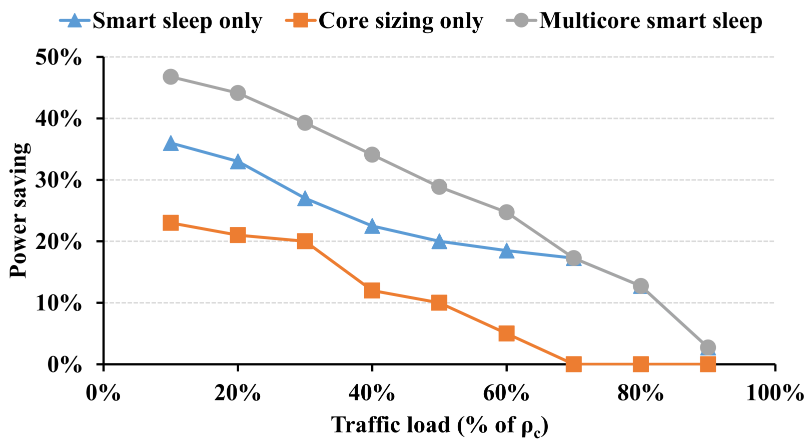


Figure 4.15: Power saving comparison among different schemes at different traffic load.

scheme always chooses the configuration with *larger* number of active cores while two or more configurations give same power consumption, as opposed to choosing smaller number in core packing [92].

Fig. 4.15 gives the power saving over default Linux scheme under our multicore smart sleep scheme with peak load $\rho_c = 0.9$. We compare the power saving results of our multicore smart sleep with two different schemes: (1) Smart sleep only, and (2) Core sizing only. In smart sleep only scheme, we do not turn off the core and only follow the principle described in section 4.3. In Core sizing only scheme, we only turn off the core without applying the smart sleep, the number of active cores is selected based on $\lceil \rho/\rho_c \rceil$. We can see that neither smart sleep only nor core sizing only scheme can efficiently exploit the opportunities for power saving. In smart sleep only scheme, although the cores will enter deep sleep during idle periods, it still suffers from state transition overheads which contribute significantly to the power consumption. On the other hand, core sizing only scheme will put the core into deepest sleep state by turning it off, but the remaining active cores still rely on default Linux C-state selection algorithm during their idle periods, resulting in

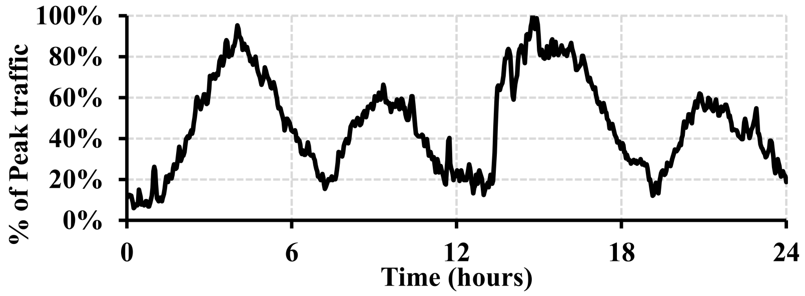


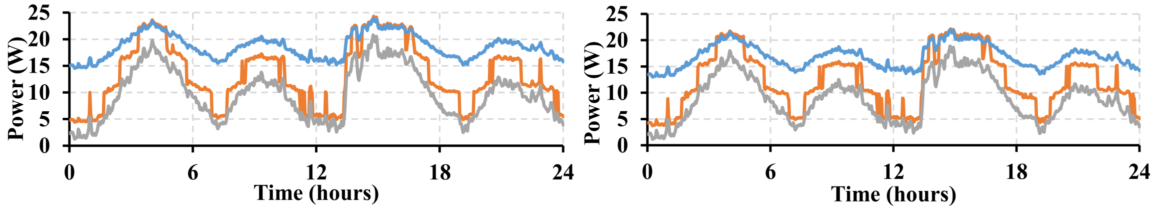
Figure 4.16: One day traffic trace recorded from CAIDA [2].

inefficient idle power consumption. Also, it will reduce the change for the active cores to enter deep sleep state, since the active cores will experience higher load. By combining both techniques, our multicore smart sleep scheme will intelligently balance among the state transition overheads, idle power savings and latency, achieving better energy efficiency.

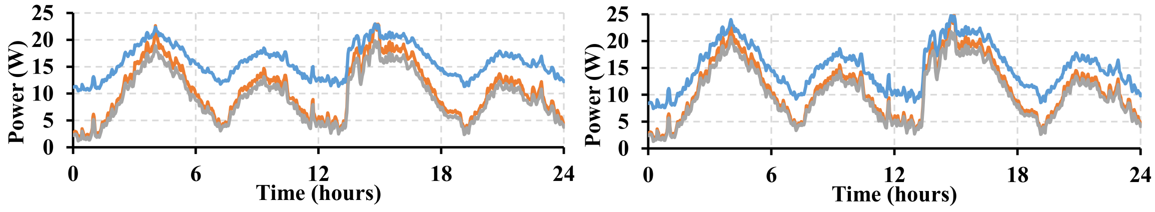
4.5.4 Runtime Results with real world traffic trace

To see the effectiveness of our multicore smart sleep scheme, we conduct experiments with real world traffic. The one day traffic trace recorded from CAIDA [2] is shown in Fig. 4.16. The traffic trace has a typical peak-valley characteristic, where the latency constraint is set at the peak load.

With the real world traffic trace, Fig. 4.17 shows the experimental results of power consumption of four applications, listed in Table 4.2. In these experiments, we scale the load so that the peak load ρ_c=0.8. Also, we show the result of (1) our multicore smart sleep scheme (MSS), (2) the default power management scheme (Linux), and (3) the theoretical perfect energy proportional scheme (EP) for the comparison. First, we can see that multicore smart sleep achieves significant power saving (38% in average) over the



(a) Memcached (b) IPV4_routing



(c) URL (d) IPSEC

Figure 4.17: Power consumption of multicore smart sleep with four different applications under real traffic trace. (peak traffic load 0.8).

default Linux scheme and close to EP. Second, the effectiveness of our scheme depends on the characteristic of applications. The power usage of our scheme is very close to the EP in *URL* and *IPSEC*, but not the other two applications. The reason is that the length of the sleep period in our scheme is dependent on the slack, which is the difference between the response time constraint and the response time at current load. For the applications with high service rate (short service time), the response time constraint is tight. The length of sleep is limited to less than the EBT of the deep sleep state. As a result, CPU core will not enter deep idle state during the sleep time and the power saving is compromised. On the other hand, the response time constraint is loose for the applications with long service time, and our scheme can achieve high power saving. Despite of the limitation, the multicore smart sleep can still achieve good power saving because of (1) dynamical core activation/deactivation and (2) elimination of C-state prediction error. In Fig. 4.17, for *memcached* and *IPv4 routing*, we can see that there are step increases/decreases of the

Table 4.3: Power saving and energy proportionality (EP) of the multicore smart sleep scheme with different applications.

	Memcached		IPv4 Routing		URL		IPSEC		Average	
ρ_c	Power Saving	EP	Power Saving	EP	Power Saving	EP	Power Saving	EP	Power Saving	EP
0.7	30.8%	0.652	33.7%	0.648	38.5%	0.717	41.2%	0.789	36.1%	0.702
0.8	31.2%	0.718	33.5%	0.723	44.3%	0.884	43.8%	0.917	38.2%	0.811
0.9	31.9%	0.819	40.3%	0.887	44.1%	0.969	42.3%	0.979	39.7%	0.914

power usage, because our scheme dynamically turns ON/OFF CPU cores based on the traffic load.

To see the effectiveness of multicore smart sleep under different QoS constraint, we repeat the experiments with different peak load, $\rho_c = 0.7$, 0.8 and 0.9. The results of power saving over the default Linux scheme are given in Table 4.3. In general, the power saving increases as c increases. This is because lager ρ_c means higher response time constraint, which helps choosing longer sleep time, entering deeper idle states and saving more power. On the other hand, for URL and IPSEC, the power saving peak at $\rho_c = 0.8$. Although our scheme can achieve significant power saving over the Linux scheme, it is not enough to show the effectiveness of our scheme. To quantify the performance, we also give the energy proportionality in Table 4.3. Energy proportionality is widely used as a performance metric for the power usage of a system, and is defined as:

$$EP = \frac{P_{MSS} - P_{EP}}{P_{MSS}} \tag{4.24}$$

, where larger EP means better power usage, and EP = 1 means perfect energy proportionality. From Table 4.3, for all applications, the energy proportionality increases as ρ_c is increased. When ρ_c is small (tight QoS constraint), the length of sleep is limited, thus the power saving is mainly brought by choosing the right size of active cores. Due to this core level granularity, the EP is limited. On the other hand, high ρ_c provides more slack for the response time, in addition to the size of active cores, long sleep with deep C-state can be applied and higher EP can be achieved. On average, the EP is 0.7, 0.81 and 0.91 for $\rho_c =$ 0.7, 0.8 and 0.9, respectively.

Chapter 5

Power Management under Tail Latency constraint

5.1 Introduction

Servers running latency-critical workloads in a data center are kept lightly loaded to meet the strict tail latency targets [21, 91]. Although servers are usually kept lightly loaded, peak load is usually considered while setting the proper target tail latency. Hence the tail latency of servers running under low load will be far lower than the target. Figure 5.1 shows the CDF of the latency of a server running under light load. We can see that there exists a "latency slack" between the tail latency of this server (dotted green) and the target tail latency (dotted red line). This latency slack provides opportunities for achieving energy proportionality.

In the previous chapters, we showed that the sleep states can efficiently reduce the

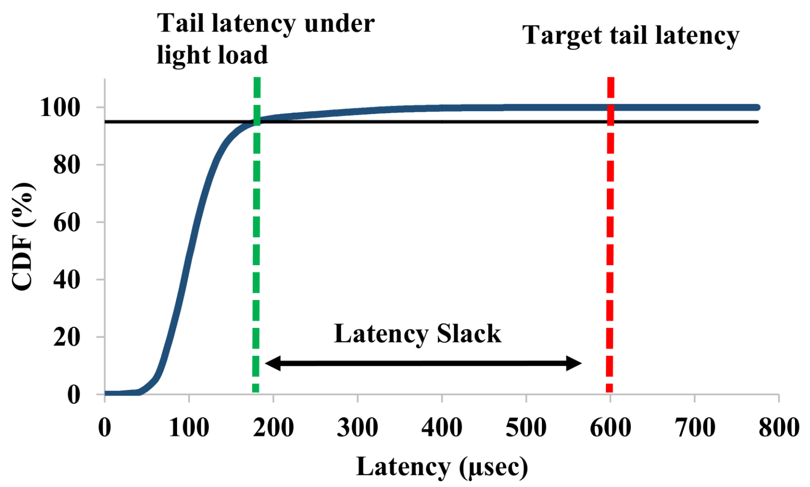


Figure 5.1: Illustration of the latency slack.

energy consumption in network and data center applications, and introduced forced sleep periods to better utilize the existing CPU sleep states. The length of the sleep periods are derived based statistically over a long epoch, assuming that the traffic behavior during the epoch is the same. In the real world datacenter workloads, the traffic exhibits high short-term variability, where request arrivals are often bursty. These short-term spikes typically dominate tail latency [23, 64, 68, 69, 86, 114] and cannot be captured with statistical models.

In this chapter, we focus on developing the power management techniques with the consideration of this traffic variability. First, we propose a dynamic sleep scheme, a fine-grain power management scheme that exploits latency slack by dynamically postponing request processing. By doing so, the idle periods that the CPU core will spend between two active periods will be prolonged, and the CPU core can efficiently enter deep sleep states to save power. Unlike previously proposed works that require some hardware changes, such as per-core DVFS with nanosecond level transition time [69] and fast sleep state transition techniques [91, 93], our proposed technique uses the commonly supported per-core sleep state.

Then, we perform a detailed analysis on the state-of-the-art techniques for both sleep and DVFS based power management scheme and find out that the existing dynamic power management techniques are largely ineffective because of the high state transition overhead, short request service time and strict tail latency constraint. Our analysis also shows that sleep or DVFS alone can not fully exploit the latency slack for power saving. We take the lessons learned here to design μDPM, an all encompassing power management framework that coordinates request delaying, per-core sleep states, voltage frequency scaling and load distribution. μDPM extends our dynamic sleep scheme by taking into the account the request processing speed. Also, we develop a novel request redirection algorithm which aims to minimize both sleep and DVFS state transitions, which are the major sources of energy inefficiency while applying sleep and DVFS power management.

5.2 Dynamic Sleep

The challenge in designing dynamic sleep schemes is in determining when the CPU core should be woken up so that the target tail latency of the requests is not violated. The traffic in datacenters exhibit short-term variability of request latency, which often dominates tail latency [64, 69]. Our scheme copes with traffic variability through novel performance modeling, which includes the requests arrival time, the statistics of the request processing time, and the queue length. This model allows us to dynamically adjust the wake up time at the request arrival instance when the CPU is sleeping, so that the target tail latency is satisfied for every request. We call this technique DynSleep

5.2.1 Overview

Figure 5.2 gives a concrete example of how the wake-up time of a CPU core is determined and adjusted in our scheme. In datacenters, the target tail latency is specified as a percentile. In this example, we use 95th percentile latency target L_C, which means that 95% of requests must be served with latency $L \leq L_C$. Assume that the CPU core finishes processing the last batch of requests and goes to sleep at $t = 0$, as shown in figure 5.2a. As time advances, the request R1 arrives at time $t = A_1$ (①), as shown in figure 5.2b. The CPU core wakes up at time $t = W_1$ (②), and the request processing time is S_1, so that the tail latency for R1 is satisfied. Figure 5.2c shows a situation when the request R2 arrives at time $t = A_2$ (③) while R1 is still waiting to be executed. The wake-up time of the core is adjusted to W_2 so that the latency of R2 is not violated (④). Similarly, when R3 arrives, the wake-up time is further adjusted to W_3. At any arrival, the core must wake up immediately if the revised waiting time falls short of the transition time of the sleep state.

In figure 5.2, no request arrives between time A_3 and W_3, thus the CPU core will wake up at W_3 and process the queued request one by one in a FIFO fashion. However, if any requests arrive after R3, the same steps are repeated until the wake-up time is met. On the other hand, when the CPU core is awake, the requests that arrive during the processing are queued. They will be processed according to the FIFO queuing principle. When the queue becomes empty, the CPU core will go to sleep, and the same procedure, described above, will be repeated. One thing to notice is that we use 95th latency target in the above equations just for illustrative purpose, it can be replaced with any latency target without changing the algorithm itself. Also, our proposed scheme supports per-core control, we only

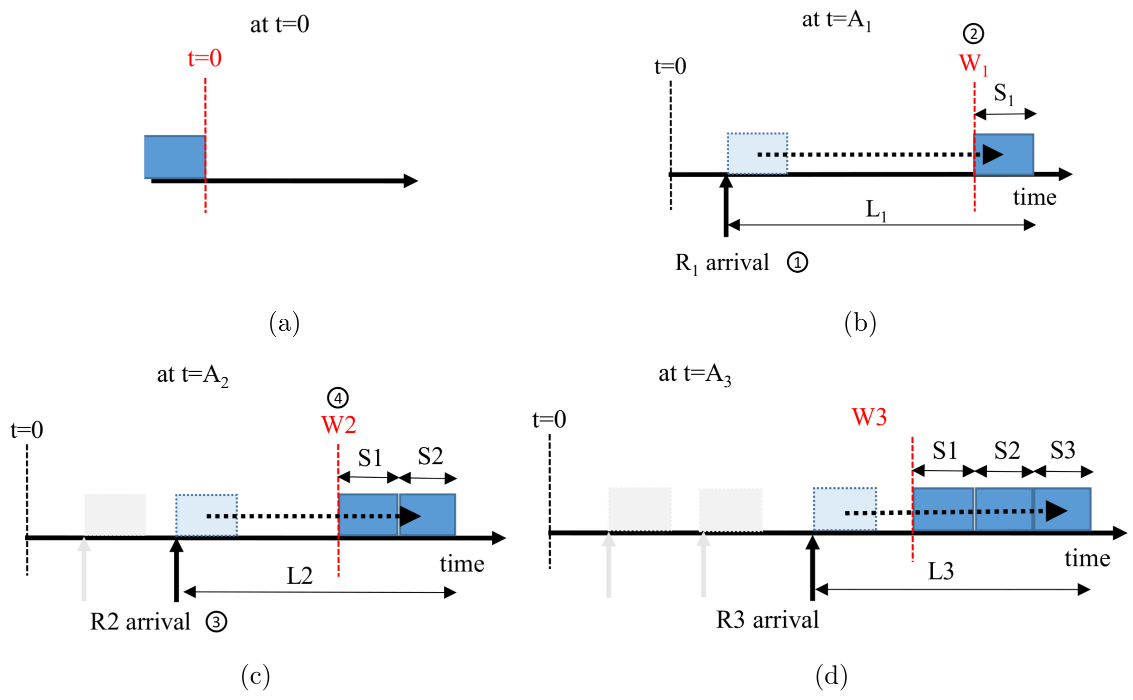


Figure 5.2: Illustration of the dynamic sleep scheme. (a)-(d) represent the different time instances as time advances.

give single core as an example. In multicore processor, every core follows the same principle independently.

5.2.2 Wake-up Time Derivation

Let S_1 be a random variable with probability distribution P_{S1}.The latency of the request of R_1, L_1, can be represented as:

$$L_1 = W_1 + S_1 - A_1 \tag{5.1}$$

With the target tail latency L_C, we can have W_1 as:

$$L_1^{95th} = W_1 - A_1 + S_1^{95th} = L_C \tag{5.2}$$

$$W_1 = L_C - S_1^{95th} + A_1 \tag{5.3}$$

, where X^{95th} means that 95% of time that the random variable X is smaller than X^{95th}. Following the same principle, we can have W_2:

$$L_2 = W_2 + S_1 + S_2 - A_2 \tag{5.4}$$

$$L_2^{95th} = L_C \tag{5.5}$$

$$W_2 = L_C - \overline{S_2}^{95th} + A_2 \tag{5.6}$$

, where $\overline{S_2} = S_1 + S_2$ is a random variable, with the probability density function $P_{\overline{S_2}} = P_{S_1} * P_{S_2}$ ($*$ is convolution). We will discuss how to compute $P_{\overline{S_i}}$ later in this section. Comparing with equation 5.3, we can see that the latency of the R_2 includes S_1, the service time of R_1, which is considered as the queuing delay for R_2. Furthermore, the wake-up time W_1 is previously determined when R_1 arrived, and is the latest time that the core can wake-up without violating the target latency, we need to compare and select the earlier time between W_2 and W_1, and update wake-up time:

$$W_2 = min(W_1, W_2) \tag{5.7}$$

By updating the new wake-up time, the tail latency of the request R_1 might change. The new wake-up time (W_2) will always be less or equal to the previous one (W_1). As a result, the new 95th latency of R_1 will become smaller than the target tail latency. Similarly, we can compute W_3 when request R_3 arrives.

Next, we compute the probability distributions by assuming the service time for each request is drawn independently from a single distribution, P_S, where S gives the service time it takes to process one request. This assumption is valid in the datacenter node due to the lack of the temporal locality in the memory system [21]. As a result, $P_{\overline{S_2}} = P_S * P_S$ and $P_{\overline{S_3}} = P_S * P_S * P_S$. In general,

$$P_{S_i} = \overbrace{P_S * P_S * P_S \cdots * P_s}^{i} \tag{5.8}$$

The service time distribution of the requests is not varying; it can easily be acquired either from the off-line analysis or a short period of learning phase. Given the P_S, P_{S_i} can be pre-computed for the future reference.

5.2.3 Implementation

We design and implement our proposed dynamic sleep with Memcached, a commonly deployed high performance keyvalue store application in the datacenter, and evaluate the power saving and latency on a real system. The experiment setup, results and analysis are given in the next section.

We use Memcached v1.4.15 [7] as our baseline. Figure 5.3 documents the process flow of memcached when processing data requests from the perspective of a memcached

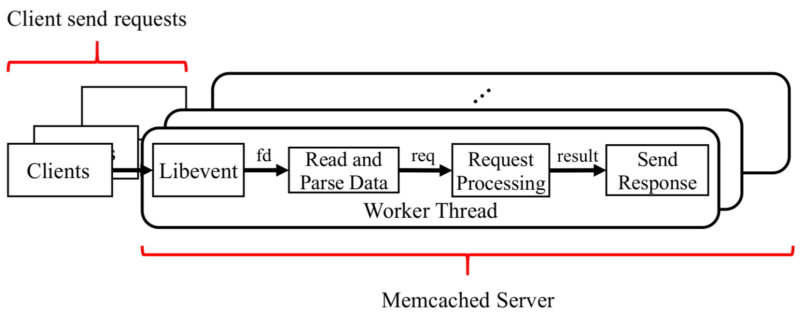


Figure 5.3: The process flow of the baseline Memcached.

server. Reviewing figure 5.3 from top to bottom, the clients are responsible for submitting operation requests to memcached. When memcached is initiated, a static number of worker threads are created to process these client requests, normally aligned with the number of physical CPU cores in the server system. Also, the mapping from the clients to the worker threads is statically built during the memcached initialization.

As shown in figure 5.3, a worker thread is initiated by *libevent*; libevent is a portable network protocol wrapper for the epoll system call in Linux, and will monitor the pre-established connections (sockets). The worker thread will wait in libevent for an event, which corresponds to the arrival of requests. When an event happens, a callback function will be called with a file descriptor (fd) as an argument to handle the event. In this case, the file descriptor is an abstract indicator of a network connection. Upon receiving an fd, the worker thread then read the data from the socket and decode the command (read and parse data). Assuming a request for a cache item, the thread computes a hash value from the key and completes a hash table lookup to identify where the value data resides associated with the key (request processing). After the data is located, it is retrieved from system memory and transmitted back to the requesting client (send response). This processing flow fits the baseline behavior we described in section 2.2, where threads are vigilantly waiting for the

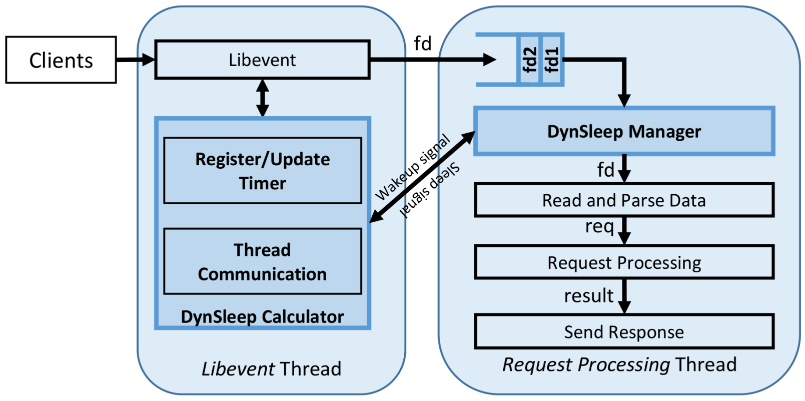


Figure 5.4: Dynamic sleep implementation of Memcached.

request, and the CPU core will wake up upon the request arrival.

Since the baseline memcached is implemented with libevent, the main challenge is to identify where and how we can apply our dynamic sleep scheme. Figure 5.4 shows the overview of our proposed memcached implementation. We separate the original worker thread into two threads: a libevent thread and a request processing thread. The libevent thread consists of the libevent wrapper and the calculator (DynSleep Calculator). The libevent will monitor the incoming request and insert the request into the queue of the request processing thread upon the request arrival. The DynSleep Calculator will calculate the wake-up time and frequency, as described in section 5.4, and register or update a timer event in the libevent through the *timerfd* API provided by the Linux. After the timer is expired, DynSleep Calculator will send a wake-up signal, through *pthread_kill*, to wake up the request processing thread. One thing to notice is that when there is an event coming, libevent only gives the corresponding socket file descriptor, indicating that there is some data ready for the read. In some cases, multiple requests are queued in the socket, and these data will be read together, similar to the batch arrival. In order to calculate the corresponding wake-up time, we need to know the number of the requests in the batch before actually reading the data. We tackle this by using *IOCTL* API to "peek" at the socket for the data size of the batch, and divide it by the average size of a request to get the number of the requests in the batch for the calculation of the wake-up time.

In the request processing thread, the DynSleep Manager is in charge of maintaining and monitoring the queue. When it sees no request present in the queue, it will notify the DynSleep Calculator in the libevent thread and go to sleep, through the sleep function. Also, upon receiving the wake-up signal, the sleep will be interrupted and the core will wake up, fetch requests from the queue and process them until the queue is empty. Note that due to the fixed connection-thread mapping, we can not implement our criticality-aware scheduler without significant application changes,

5.2.4 Evaluation

Experiment setup

We use two multicore machines, one client server and one request processing server, connected over Ethernet. The client server will establish multiple TCP connections with the request processing server, emulating multiple clients sending requests. The inter-request time for each client follows exponential distribution, and is scaled to see the impact of different traffic loads on the request processing server.

We are mainly interested in the behavior of the request processing server with an Intel Xeon E5 2697-V2 12-core processor. In our experiments, Hyperthreading and Turbo Boost are disabled. The per-core frequency can be adjusted independently through clock modulation [4]. However, different from per-core DVFS that requires per-core voltage regulator, this processor use a off-chip voltage regulator, hence the voltage across all cores must stay the same. The sleep state characteristics are given in Table 5.1. Also, this Intel processor is equipped with on-chip energy sensors for CPU cores. Through the MSR, the power consumption results are collected at the rate of 1 KHz.

In the request processing server, we run 3 instances of Memcached simultaneously, each with 4 threads. This is a recommended workaround to a known scalability issue with Memcached [7]. Also, the thread-to-core mapping is fixed to avoid run-to-run variance caused by the Linux thread scheduler [84]. For our version of Memcached, all libevent threads are running on one core, and request processing threads are mapped one-to-one to the CPU cores. We define tail latency as the 95th percentile latency; the latency target is set at the tail latency of the baseline Memcached running at 80% load. We measure the request latency through timestamping the request and response at the client server, which includes the TCP stack processing time from both client and server and the network latency.

5.2.5 Results

In this paper, we limit our evaluation on practical real-world implementations, hence did not compare with other work. In Rubik [69], authors assume that there exists a fast per-core DVFS mechanism that has transition latency in few nanoseconds. However, most of the existing servers have about 100 micro second DVFS transition overhead due to the off-chip voltage regulator, which greatly impacts the performance of Rubik, especially in the applications with micro second level service time; In SleepScale [85], the computation overhead for the optimum configuration is too long (>5ms in our experiment) which leads to severe tail latency constraint violation. PowerNap or DreamWeaver [91, 93] are based on hypothetical servers which can transition into full-server sleep states in the order of microseconds. Furthermore, we observe that Memcached idle periods are too short for PowerNap to exploit, hence resulting in *worse* energy consumption due to the state transition overhead.

To show the effectiveness of our dynamic sleep scheme, we compare our results with four power management schemes: Baseline, DVFS, DFS and EP schemes. In the baseline scheme, frequency of CPU cores is set to be maximum at all time. In DVFS scheme, we monitor the latency over an epoch, and then adjust the voltage-frequency configuration at the end of the epoch[1], similar to Pegasus [86]. The frequency can be set in the range between 1.2 and 2.7GHz with 100MHz steps. Since our experimental platform doesn't support per-core DVFS, we estimate DVFS savings by running the application with single thread on one core, and turn off all other cores. The power consumption results are then scaled by the number of cores in the CPU, assuming all cores will perform identically. The DFS scheme is similar to DVFS scheme, except that we only change the frequency for each core, the voltage is kept the same. The EP scheme is the idealized power saving, which assume the power consumption is perfectly proportional to the load.

Power saving

Figure 5.5 reports the average power saving over the baseline scheme for four different power management schemes at different CPU loads. DynSleep performs the best

[1]We set the epoch as 50 ms to accommodate the DVFS transition overhead.

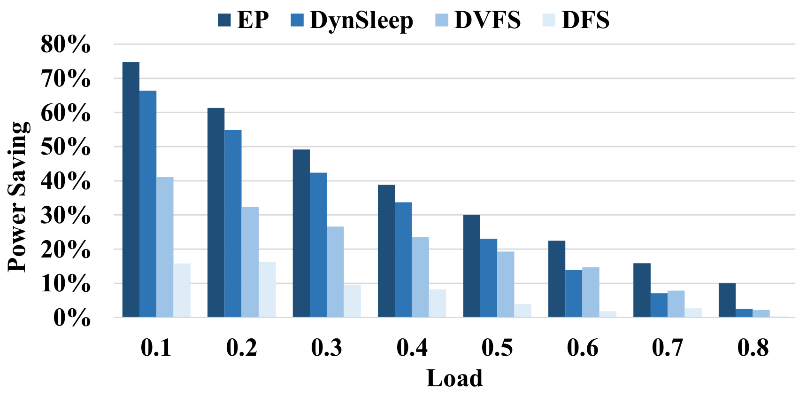


Figure 5.5: Power saving comparison among different power management schemes.

at low to medium load (10 50%) among all schemes (except for EP), and is comparable to DVFS scheme under high load. At low to medium load, because of the larger latency slack, DynSleep can efficiently create longer idle periods for the CPU cores to enter deep sleep states. However, as the load increases, the latency slack and idle periods decrease, and DynSleep can only drive CPU cores into shallow sleep states. It is important to note that the DVFS scheme is not supported in most of commodity servers due to the lack of per-core voltage control. A more realistic scheme is the DFS. DynSleep significantly outperforms the DFS scheme at all cases. Moreover, the power saving achieved by DynSleep falls within 10% of the EP on average. Because of the sleep state transition overheads, CPU cores must spend some time and power on entering/exiting sleep states, preventing DynSleep from achieving perfect energy-proportionality.

Shifting Latency Distribution

Figure 5.6 shows the latency distribution of the baseline, DFS (also represents DVFS[2] and DynSleep scheme under a load of 0.3. In these experiments, we run each

[2]The latency distribution of DVFS and DFS are similar, because the request processing time is affected by the frequency only.

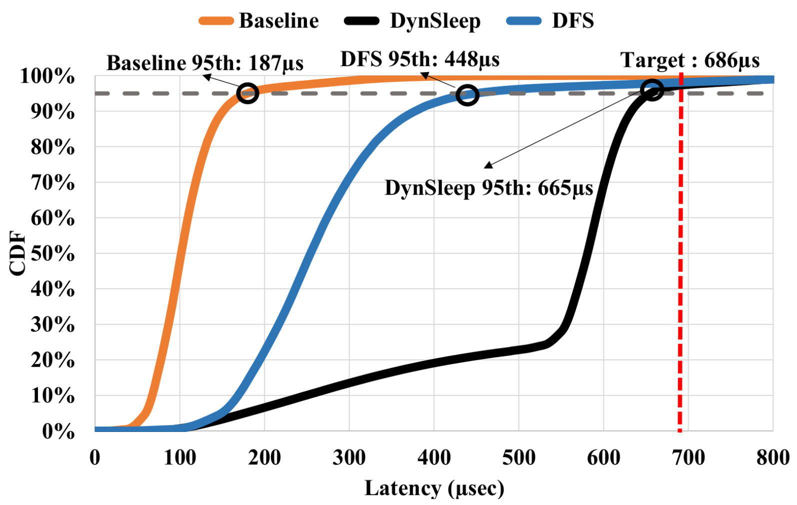


Figure 5.6: Latency distribution and tail latency under different power management schemes.

scheme for one minute with the fixed load. The tail latency under each scheme is circled, and the target tail latency is given in dashed red line. We can see that although there exists a large latency slack, about 500 sec between baseline and target tail latency, the DFS scheme, with tail latency 448 sec, cannot fully utilize this slack. DFS scheme slows down the request processing, hence increasing the service time and queuing delay. However, the dynamic range of the frequency scaling limits the effectiveness of the DFS on exploiting the latency slack and power saving. Our DynSleep scheme further exploits these opportunities by dynamically postponing the request processing, and leads to a better use of this slack and power saving. One thing to notice is the sharp knee of the curve of the CDF of the latency in our DynSleep. The slow slope before the knee is caused because some requests arrive during previous request processing and the burst of incoming traffic. The requests that come during the processing period will not experience the additional delay caused by CPU sleeping, it will simply wait until all previously queued requests finish their processing.

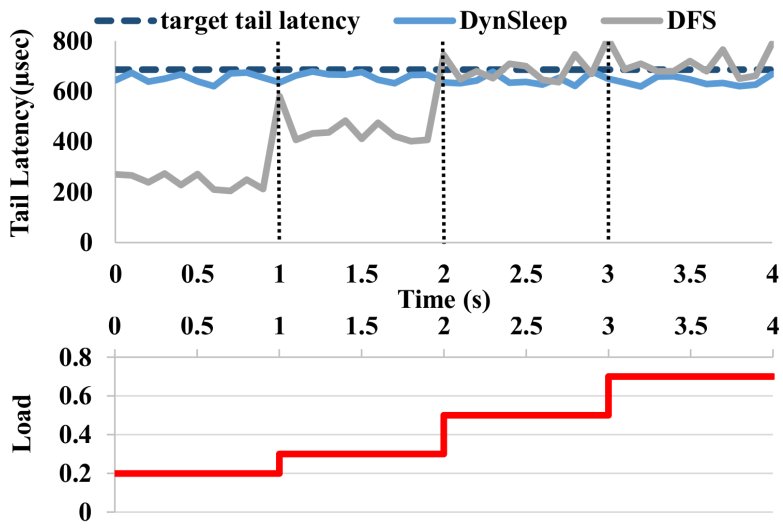


Figure 5.7: The tail latency of different power management scheme and different load.

On the other hand, during the bursts of traffic, the wake-up time is mostly determined by the request coming latter (ex: R3 in figure 5.2). As a result, the waiting times of queued requests are shorter, and their latency will fall far below the target. The high slope after the knee is caused by the requests that are queued and waiting due to DynSleep. As can be seen, DynSleep is able to shift a significant portion of the request latency distribution towards the target tail latency without violating it.

Sensitivity to Load Changes

In addition to meeting the target tail latency in the steady state (figure 5.6), DynSleeps design allows it to respond instantaneously to sudden changes in input load. Since the wake-up time is updated at every request arrival instance, DynSleep reacts to the changes in the inter-request arrival time immediately. Figure 5.7 shows how DynSleep and DFS respond to sudden load change. The input load is shown in the bottom, which grows in steps over a span of 4 seconds, from 0.2, 0.3, 0.5 to 0.7. The 95th percentile latency for two schemes are shown in the top over a 100ms window. First, we can see that the DFS scheme will lead to a latency spike at the time of load changes, as well as during high load (0.5 and 0.7). This is caused by the high granularity of epochs. The frequency can only be changed at the end of an epoch. As a result, DFS is lacking the responsiveness to the load change, and suffering frequent latency violations. Also, at low load, 0.2 and 0.3, the limit of the frequency scaling yields unnecessarily low tail latencies and wasting power. By contrast, DynSleep is able to achieve stable tail latencies under all loads. At low load, DynSleep chooses to wake up the core later, and achieves tail latency close to the latency bound, saving additional power. Also, it adapts quickly to the sudden load changes, thanks to the dynamic wake-up time adjustments.

5.3 Implications of microsecond request service time on dynamic power management

In this section, we will explore the implications of microsecond request service time on dynamic power management policies. A guiding principle in microservices architecture is to decompose applications into smaller, interconnected services. For example, a typical e-commerce website may consist of various microservices, such as a web server front end; various business logic services for product ordering, accounts, and inventory; databases and data stores, such as Redis, Memcached, and MongoDB; and other specialized functionalities, such as search and recommendation systems.

We explore various state-of-the-art dynamic power management schemes, such as a VFS-only technique (Rubik [69]), a coordinated VFS and Sleep technique (SleepScale [85]), and a Deep Sleep-only technique (DynSleep [33]). All of these prior techniques exploit existing latency slack and are quality-of-service aware to meet tail latency constraints. Power savings can be achieved by slowing down request processing through VFS [69, 85, 86, 114] or delaying request processing through Sleep [33]. Due to it's better responsiveness in handling short-term variation, we will evaluate Rubik as representative of other VFS techniques, such as Pegasus [86] and TimeTrader [114]. While other Deep Sleep-only techniques exist, such as PowerNap [91] and DreamWeaver [93], we observe limited power savings due to the requirement for full-system idleness where the entire processor socket is idle. CARB [121] also leverages sleep states in latency critical applications by selecting the optimal number of cores to keep on, placing inactive cores into deep sleep. However, active core can still consume significant idle power due to shallow sleep state selection. Therefore, we evaluate DynSleep [32] as the target Deep Sleep-only mechanism. Unlike PowerNap and DreamWeaver, DynSleep is able to achieve power savings through sleep by leveraging fine-grain per-core sleep states, rather than whole socket sleep states.

To identify the effect of microsecond request service time on dynamic power management, we evaluate four workloads representing common latency-sensitive microservices. More workload details will be discussed in section 5.4.5. We run these workloads on a server with Intel Xeon E5 2697-V2 12-core processor. We will now highlight various observations and implications to dynamic power management.

Observation 1: The majority of request service times are less than ~250μs, even with millisecond tail latencies[3]

[3]Carefully note we make a distinction between *service time* and *latency*. Latency includes queueing time and service time.

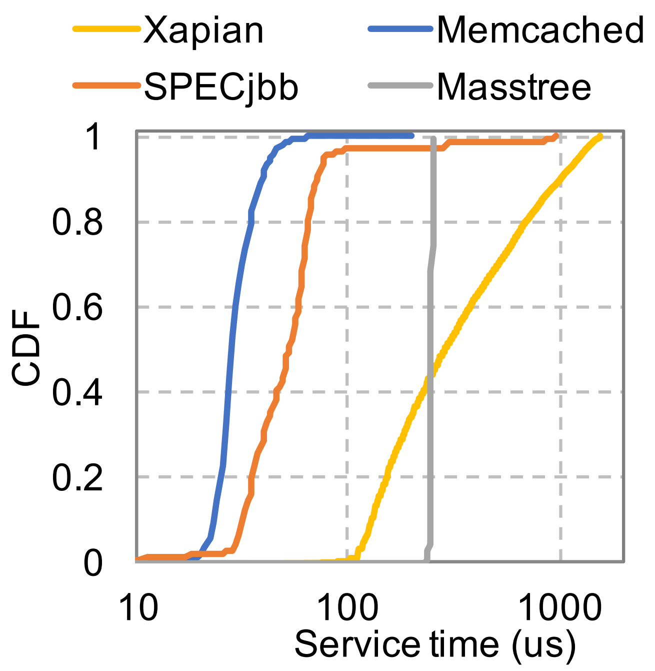


Figure 5.8: Service time dist. Service times are commonly <250μs.

We define the tail latency similar to [69] as the 95$^{\text{th}}$ percentile running at medium load. The target tail latency of these workloads are 150μs for Memcached, 800μs for SPECjbb, 1100μs for Masstree, and 2100μs for Xapian. Despite the length of the target tail latency, the actual typical request service time is significantly shorter and often an order-of-magnitude shorter.

Figure 5.8 shows the observed request service time distribution. In our experiments, we observe that 95% of request service time complete within 33μs for Memcached, 78μs for SPECjbb, 250μs for Masstree, and 1200μs for Xapian. Despite Xapian's long service time tail, a significant 45% of Xapian's request still completes under 250μs. These short service time requests dominates, and can lead to high short-term variability in load, making DPM especially challenging [69].

Observation 2: Dynamic power management breaks down at microsecond request service times

Figure 5.9 shows the average power consumption of various state-of-the-art dynamic power management techniques across different request service times. For this experiment, we simulated an exponential service distribution with varying average service time, shown on the x-axis. The synthetic workload was driven at a medium load. We ran this experiment across a range of server loads, and observed similar trends. In the baseline scheme, the processor operates at maximum frequency, and uses the Linux menu governor [103] to choose the C-state (sleep state) during idle periods. We similarly use the Linux menu governor to manage sleep states for Rubik (VFS-only), with frequency states updated at every incoming

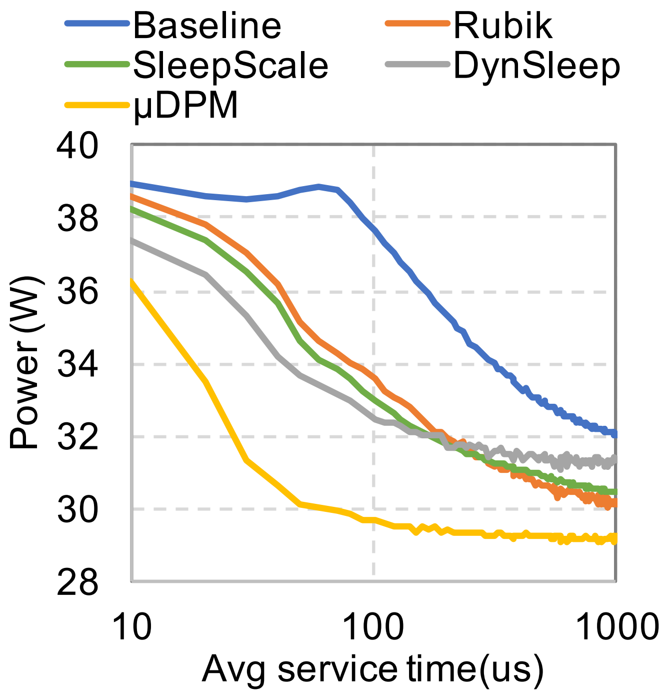


Figure 5.9: Power consumption. In this short service time region, state-of-the-art power management schemes has diminishing effectiveness. Counter-intuitively, deep sleep states provide the most benefits in this region.

request. SleepScale (VFS+Sleep) selects the optimal C-state and optimal frequency based on historical profiling. DynSleep (Deep Sleep) directly enters C6, the deepest sleep state, and runs at maximum frequency.

In general, as the request service time decreases, power increases due to less opportunities for low power states. In the 250-1000μs range, we observe that dynamic power management schemes that utilize VFS provides the lowest power due to having ample opportunity to slow down request processing to save power. However, once the average service request times drops below 250μs, both VFS-based and Sleep-based techniques begin to breakdown. This is because VFS techniques cannot handle the short-term variability of short request service times and cannot find enough opportunities to slow down request processing. Meanwhile, Sleep techniques cannot enter a deep enough sleep state due to short idle cycles. Surprisingly, techniques utilizing Sleep begin to outperform techniques utilizing VFS! This can be explained by the trend in idle period lengths.

Observation 3: Short service times fragment idle periods

It is well known that *utilization* have a significant impact on sleep opportunities [91, 93]. We additionally find that *request service time* also have a significant impact on sleep opportunities. Specifically, short request service time can fragment idle periods into short idle periods that sleep states cannot take advantage of.

Figure 5.10 shows the idle period length (in μs) under 200μs service time (dotted line) and 500μs service time (solid line). Clearly, as the service time decreases, the idle period lengths similarly decreases. The ever-decreasing idle period length can eventually lead to the ineffectiveness of low power states. For example, the baseline curve (based on

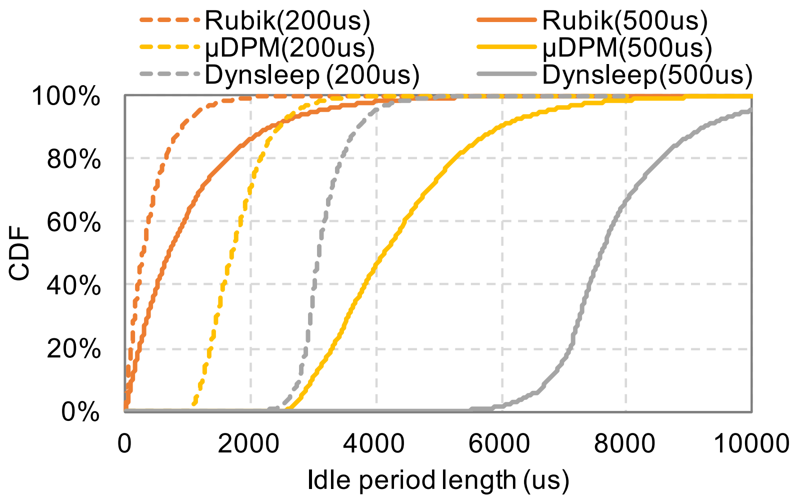


Figure 5.10: Idle period length. Short request service time causes idle periods to become fragmented.

Linux menu governor) in Figure 5.9 can no longer save any power at around 80μs service time due to the inability to enter sleep state.

In figure 5.10, we also observe that the dynamic power management scheme used can also have a great effect on the idle period length. Specifically, Sleep-based techniques are able to significantly extend idle period lengths, even under very short service times. This is due to the ability for sleep states to consolidate idleness by delaying request processing. By coalescing idle periods into longer idle periods, sleep states are better utilized.

Observation 4: State-of-the-art dynamic power management suffers from significant transition and/or idle power

Existing state-of-the-art power management technique has focused primarily on exploiting latency slack to save power, with little consideration of the energy overheads due to state transitions and idleness. In our experiments, we found that the energy overheads of state transition and idleness account for a significant fraction of energy consumption.

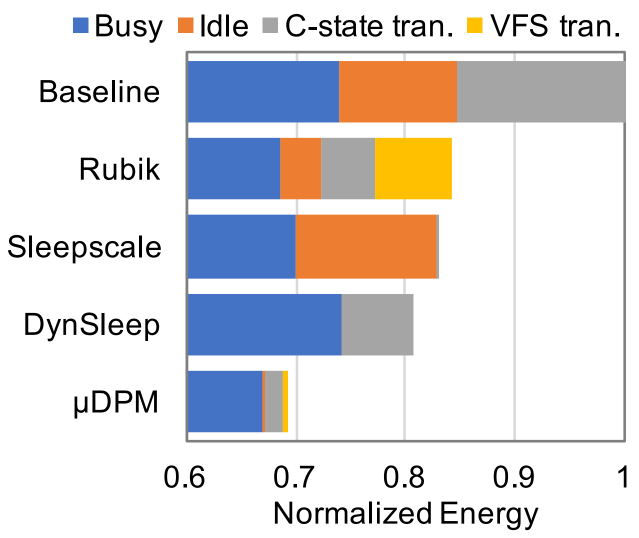


Figure 5.11: Energy consumption breakdown. Significant energy is wasted due to idle and transition overheads.

Figure 5.11 shows the energy consumption breakdown of state-of-the-art power management techniques. For this experiment, we ran a synthetic workload with average service time 80 μs. In this figure, total energy consumption is broken down into 4 parts: (1) energy to process requests (*busy*), (2) energy consumed when a core is idle—can also be in a shallow C-state—and waiting for requests(*idle*), (3) the energy spent on C-state transitions (*C-state tran.*) and (4) the energy spent on VFS transitions (*VFS tran.*). The energy consumption are normalized to the total energy consumption of baseline.

Baseline, Rubik and SleepScale all have significant idle energy due to sub-optimal sleep state selection, which reflects our previous observations. Idle energy accounts for 5% of total energy consumption in Rubik and 15% in SleepScale. Due to DynSleep always entering C6 state when encountering idle periods, the idle energy consumption is virtually non-existent. However, DynSleep suffers from significant C-state transition overhead, accounting for 18% of total energy consumption. SleepScale limits power state configuration to every epoch, which limits transition overheads to only 5% of total energy consumption. Due to Rubik's aggressive VFS reconfiguration at every request arrival, VFS transition overhead accounts for 10% of total energy consumption. Prior work [69] and our experiments observe VFS transition latency between 100~130 μs. Despite the existence of fast integrated on-chip voltage regulator, the internal microcontroller of the Power Control Unit [28] spends significant time managing these voltage regulators, resulting in long transition latency. Because of this long transition latency, care must be taken while designing power management scheme with VFS.

Fallacy: Tail latency is more important than average latency when designing dynamic power management.

While tail latency is a direct measure of quality-of-service and the central focus of many dynamic power management policies, we demonstrated here that average service time is also very important in enabling energy efficiency. As shown before, short service times can cause shorter idle period lengths, which hampers sleep states, and can also lead to higher short-term variation, which hampers frequency scaling. The order-of-magnitude difference between the service time and the target tail latency provides significant opportunity for dynamic power management, highlighting the need for DPM to be aware of service times. As we will later see, we can leverage this property for energy efficiency by increasing the average latency while keeping the tail latency the same.

Fallacy: Sleep states are too slow to be used in latency-critical workloads

It has been widely assumed that Sleep states are not applicable under strict tail latency constraints [86], as previously shown in the baseline case. Surprisingly, we found that deep sleep modes can be beneficial as long as it is tail latency aware. Even more surprisingly is that tail latency-aware deep sleep states actually work better than VFS-based techniques for service times under 250μs. DynSleep is able to directly enter C6 state, which consumes ~0W when the CPU core is idle and delays request processing as long as tail latency constraints are met. This request delaying behavior leads to superior idle period lengths, as shown in Figure 5.10. We find the idle periods for DynSleep with 200μs service time is at least double that of Rubik at 500μs service time, enabling ample opportunities for sleep states in latency-critical scenarios.

Note however, that longer idle period length does not always translate to better power savings. At service times above 250μs, Rubik outperforms DynSleep due to the benefits of VFS in slowing down requests. Essentially, with short request service time, VFS cannot *extend* request processing time, but Deep Sleep can *delay* request processing time, in order to exploit latency slack for power savings. *Therefore, the key to dynamic power management with millisecond service time is to delay request processing through deep sleep modes.*

Table 5.1 shows the empirically measured idle state power and transition overhead from our evaluation server, along with the target residency time. CPU cores consume significantly lower power in deeper C-states, along with increasing transition overheads since it takes more time to disable/enable the corresponding on-chip components. During state transitions, the core consumes full power. Thus, the CPU core should only enter a particular C-state only if the idle duration is greater than a threshold, called the target residency time [5], in order to result in net energy savings. Therefore, C1 is optimal with

Table 5.1: Measured CPU core C-states on Intel Xeon E5-2697 v2 processor.

State	State tran. time	Residency time	Power per core
C0	N/A	N/A	∼4W
C1	1μs	1μs	1.43W
C3	59μs	156μs	0.43W
C6	89μs	300μs	∼0W

idle period length of 1-156 μs, C3 is optimal with idle period length of 156-300 μs, and C6 is optimal when idle period length is beyond 300 μs.The Linux idle governor estimates the idle period length and selects the best state for that idle period. If idle periods are short, as is common with short request service times, then the governor will consistently select shallow sleep states. SleepScale's sleep policy selects the best sleep state given the previous epoch's idle period properties, and can similarly select shallow sleep states consistently under short request service times. As shown in Figure 5.10, deep sleep techniques creates idle period lengths that are amenable to C6 states. Therefore, to avoid missed power savings opportunity, we should directly enter C6 state upon encountering an idle period, delay requests, and coordinate with VFS to meet tail latency constraints.

Summary: Microsecond request service time pose many unique and difficult challenges for existing dynamic power management schemes. Many microsecond-era workloads observe extremely short request service times, with 95% of requests under 250μs, with tail latencies typically in the millisecond range. Existing dynamic power management techniques are largely ineffective in this range. Compounding this, existing techniques also suffer from significant energy overheads (up to 20%) due to power state transition overheads and idleness. Interestingly enough, we observe that deep sleep states can hold the key to saving power in this microsecond service time (<250μs) range. Clearly, no prior work provides an all-encompassing solution to meet strict SLA requirements, minimize idle energy and state transition overhead energy in the microservices era. We take the lessons learned here to design μDPM. As can be seen in figure 5.8-5.11, μDPM is able to effectively coalesce idle periods and avoid transition overheads, leading to significantly more power savings than state-of-the-art DPM under extremely short service times.

5.4 μDPM

In this section, we propose μDPM, a dynamic power management scheme for the microservices era. μDPM solves all of the aforementioned limitations by coordinating deep sleep, VFS, and request delaying to fully exploit latency slack. This is achieved by postponing request processing, by delaying and slowing down, in order to finish just-in-time to meet the target tail latency. We will first give an overview of our proposed power management framework followed by a detailed description and an example.

5.4.1 Overview

The main idea behind μDPM is to aggressively deep sleep, and coordinate sleep length and VFS to delay and slow down request processing in a tail latency-aware manner.

The moment a core is idle, μDPM immediately goes into the deepest sleep state (C6) in order to save idle power. By prolonging the idle period between two active periods, we forcibly coalesce short idle periods created by short request service times, so that the core can efficiently enter deep sleep states to save power.

Upon wakeup, μDPM will operate at the lowest frequency that will satisfy the queued requests' target tail latency. Instead of waking up the core at the latest moment and processing at full speed to meet tail latency targets, we wake up the core earlier and process the request at a slower rate using VFS to achieve a better trade-off between latency and power savings. Then as new requests arrive in the system, we will update the frequency as needed in order to satisfy the target tail latency of all requests in the system. We will refer to requests that trigger a frequency or wake-up time change as a *critical* request. Critical requests are requests which will violate the tail latency under current wake-up time and frequency configurations.

In order for μDPM to work, we need to answer: When to wake up after sleeping? and What frequency to run at? The key to answering these questions lie in being able to estimate the incoming request's service time. This is especially challenging in data centers due to short request service time, which cause significant short-term variability that often dominates tail latency [64, 69]. To account for this, we developed a novel performance model that takes into account traffic variation. μDPM will recalculate wake-up time and frequency at every request arrival when the core is in sleep mode. In addition, μDPM will also carefully takes state transition overheads into account while determining the optimal wake-up time and runtime frequency. The exact design of our performance model is given in section 5.4.2.

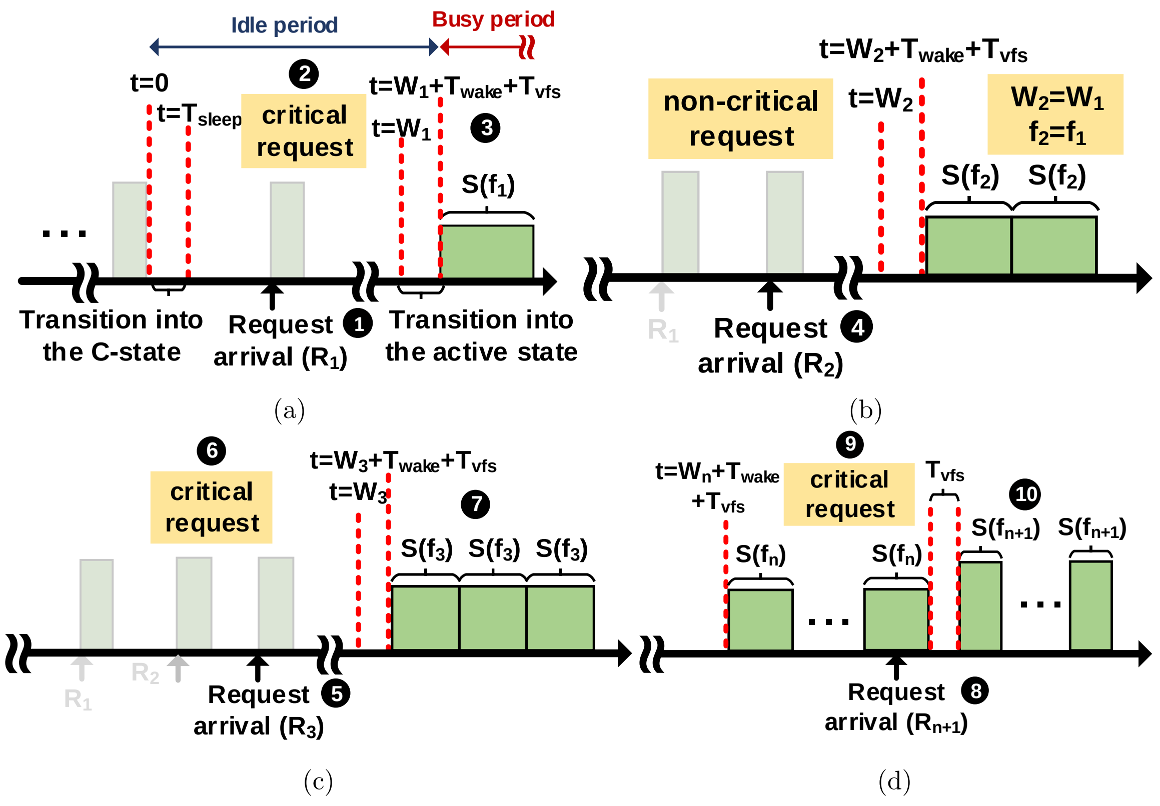


Figure 5.12: Illustrative example of μDPM. (a)-(d) represent the different scenarios as time advances: (a) the core become idle, and then a request arrives, (b) a non-critical request arrives, (c) a critical request arrives triggering configuration change, (d) during active request processing, a critical request arrives triggering frequency reconfiguration.

μDPM Walkthrough:

We will now present an illustrative working example of how μDPM works. Figure 5.12 presents how the wake-up time and frequency of a processor core is determined and adjusted by μDPM. For conciseness, we will refer to the wake up time and frequency pair as a *configuration*. Assume that the processor core finishes processing the last batch of requests and goes to sleep at $t = 0$, as shown in figure 5.12a. The core will then spend some time T_{sleep} transiting into the sleep state while consuming full power. Upon sleeping, the wake up time is cleared, meaning that it won't wake up until further configuration updates (i.e. when a request arrives).

As time advances, the first request R_1 arrives at time A_1 (❶), as shown in 5.12a. Since the wake-up time is cleared and the processor is not scheduled to wake up, request R_1 will definitely violate the latency constraint and will be identified as critical (❷). The configuration computation is then triggered to determine the new wake-up time (W_1) and frequency (f_1) for request processing (❸), and the request is inserted into the queue. Note that the request will not immediately wake up in order to maximize power savings in deep sleep mode. Upon wake-up, the core will spend time T_{wake} transiting from sleep state to active state at full power. After that time, T_{vfs} will be spent on waiting for the voltage and frequency scaling to become stable. As a result, the processor core will be ready to process at time $W_1 + T_{wake} + T_{vfs}$. The tail service time for the request is $S\left(f\right)$, which is a function of the operating frequency. The time W_1 is set such that request R_1 will complete as close to the target tail latency as possible ($W_1 + T_{wake} + T_{vfs} + S\left(f_1\right) - A_1 \rightarrow$ target tail latency).

Figure 5.12b shows how μDPM handles multiple pending requests. Here request R_2 arrives at time A_2 (❹) while R_1 is still waiting to be executed. Assume R_2 is not identified as critical, therefore the configuration is not updated and the request is simply inserted into the queue directly. Request R_2 is not considered critical as long as the elapsed time between the arrivals of R_1 and R_2 is greater than the tail service time. Intuitively, when the processor wakes up and process R_1 just-in-time, R_2 will be processed right afterwards before R_2's target SLA is reached. If R_2 arrives closer to R_1, then when R_2 is processed, it will end up violating the target SLA.

Figure 5.12c shows the arrival of the third request R_3 at $t = A_3$ (❺). Since the arrival time between R_2 and R_3 is shorter than the tail service time, R_3 will be critical and the configuration needs to be re-computed(❻). New configuration (W_3, f_3) is then determined and updated such that R_3, the critical request, will complete close to the target tail latency(❼). The process will continue until the timer expires at $t = W_n$. Then the core wakes up and begins processing requests at $t = W_n + T_{wake} + T_{vfs}$ with frequency f_n, as shown in figure 5.12d. For brevity, we don't show request arrivals between A_3 and W_n.

As we mentioned previously, the requests that arrive during processing might violate the latency constraints when the CPU core is awake and busy. As shown in figure 5.12d, the request R_{n+1} arrives at $t = A_{n+1}$(❽) when the core is processing requests. R_{n+1} will be processed after all the previous requests are processed. In this example, R_{n+1} is critical(❾) and frequency update can be computed to avoid latency constraint violation (❿). The frequency transition overhead is considered while computing the update. Note that we do not reduce the frequency during the busy period. We allow only frequency increase to limit transition overhead. This further increases the idle period and avoids any possible extension to the busy period. Every request arrival during busy periods will be checked to adjust the frequency, and the core will become idle again after the queue is emptied, as shown in figure 5.12a. While our proposed scheme supports per-core control, we only illustrate single core as an example in figure 5.12. In multicore processor, every core follows the same operation independently.

5.4.2 Performance Model

We will now present our statistical performance model to show how request service times and latencies are estimated, when to trigger new wake up and frequency configurations, and how configuration updates are computed. Figure 5.14 illustrates the various components of our performance model.

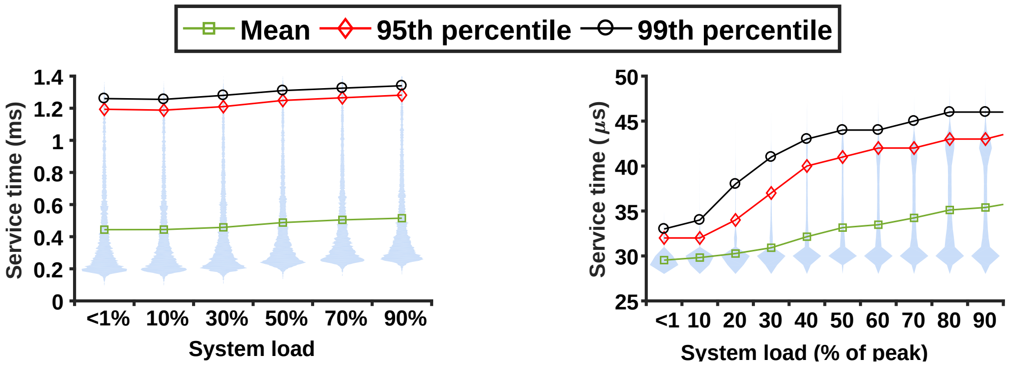
Estimating Request Service Time: Because of the non-deterministic request demands, the service time of a request is often considered as a random variable. Previous work assumes that the service time for each request is drawn independently from a single distribution [21]. While it has been shown previously that different request types can also have different distributions [57], we choose to utilize a single distribution for simplicity, trading off a small amount of power savings opportunity.

Considering the effect of frequency scaling, the service time of a request is further broken down into two parts. (1) The time (clock cycles) spent in the core, X, when executing the instruction and stalling on private caches. We use clock cycles instead of the wall clock time since it will not be affected by different VFS states. (2) The time (wall clock) spent on share resources, M, such as stalls on LLC and main memory access. Both X and M can be seen as random variables. The service time under different frequency, S_f, can be modeled as:

$$S_f = \frac{X}{f} + M \tag{5.9}$$

, where X and M for each request are independent random variables. Prior work assumes that these random variable are drawn independently from computation and memory-bound time distributions [69]. However, this model does not capture the performance impact of shared resources existing in multicore processor [87, 123], which affects M.

In Figure 5.13 we demonstrate the need to capture performance impact of shared resources. The figure shows experimental results of the service time distribution under



(a) Xapian (b) Memcached

Figure 5.13: Service time variations under different system load. The shaded areas are the PDF under each scenario with the width representing the probability at certain service time.

different system load with fixed operating frequency for multi-threaded Xapian (search engine), and Memcached (key-value store) applications. The details of the applications are given in section 5.4.5. In these experiments, we run two processes, a record process and stress process, for each application. The record process run the single-threaded application to log the service time with less than 1% overhead, and is pinned on a fixed core. On the other hand, the stress processes run multiple threads, and the load is varied to achieve different total system load. Note that the record and stress processes are running the same application. The purpose of this experiment setup is to avoid the service time variation caused by shared data locks [84,102]. In figure 5.13, we can clearly see that the service time distribution is highly depending on the total system load, as well as the application itself. As load increases, the stalls on LLC and main memory access will also increases.

Considering this load dependent service time variation, we revise the service time estimation as:

$$S_f = \frac{X}{f} + M(\rho) \tag{5.10}$$

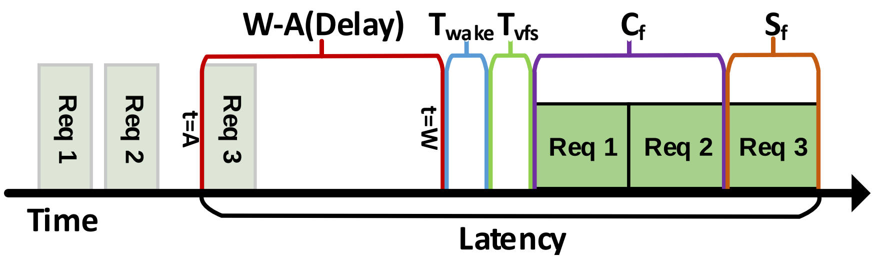


Figure 5.14: μDPM performance model

, where ρ is the load.

The distribution of X and $M(\rho)$ can be acquired through off-line profiling and can be periodically updated with on-line profiling using performance counters and CPI stacks [45, 69, 97].

Estimating Request Latency during Idle Period: Once we estimate the request service time, we then need to estimate the request's latency. Figure 5.14 gives the breakdown of the latency under μDPM. The latency (L) for a request arriving at time A during an idle period can be represented as:

$$L = W - A + T_{wake} + T_{vfs} + C_f + S_f \tag{5.11}$$

The wake-up time W and frequency f are the previously determined wake up/VFS configurations before this request arrival, T_{wake} is the wake-up transition overhead, T_{vfs} is the VFS transition overhead and C_f is the time to process the previously arrived requests. We assume each request is independent, and therefore, C_f can be seen as a random variable drawn from a distribution formed from the n-fold convolution of requests' service time distribution, where n is the number of previously arrived requests. For example, in figure 5.14, the latency model of the 3rd request calculates C_f as a convolution of two S_f. If another request arrives, that 4th request will calculate C_f with a convolution of three S_f and so on. By considering the request arrival time (A) and the time to process previously arrived requests (C_f), our latency model can effectively capture the traffic dynamics and accurately estimate the request latency.

Determining Critical Requests After estimating the latency of arriving requests, we need to check whether or not the arriving request is critical to determine whether a configuration update is needed. By observing figure 5.14 and equation (5.11), a request is critical if

$$A - A_{prev} \leq S_f^{tail} \tag{5.12}$$

, where A_{prev} is the arrival time of the previous request. This is because the arrival time between the current and previous request is too short for processing one request. Intuitively, for a given wake up/VFS configuration, when the processor wakes up, it can process a request every S_f seconds and meet the target tail latency just in time. If requests arrive too close together, since the previous request is scheduled to finish just in time to meet the target tail latency, the current request will experience longer latency than the previous one, exceeding the tail latency target.

Determining New Wake-Up Time and Frequency Configuration. In order to determine frequency configuration upon wake-up, we use the following energy model. The energy consumption for a given configuration (wake-up time W and frequency f) is:

$$\begin{aligned} E(W, f) = {} & (W - T_{sleep}) P_{idle} + (T_{sleep} + T_{wake}) P_{max} \\ & + T_{vfs} P_{vfs} + \left(C_f^{avg} + S_f^{avg} \right) P_f \end{aligned} \tag{5.13}$$

The first term in equation (5.13) is the idle period energy with the idle power P_{idle} and sleep transition time T_{sleep}. The second term is the transition energy overhead of going into and out of sleep mode. The third term is the voltage and frequency transition energy, and the last term is the busy period energy consumption with average queuing delay and average service time multiply the active power P_f. Note that we assume the idle period started at $t = 0$. The configuration update is then to compute a new configuration (W, f), which minimizes the energy consumption while maintaining the tail latency constraint. Although the configuration can be mathematically derived with some simple power model, e.g. $P_f \propto f^3$ or $f^{2.4}$, the real relationship between frequency and active power consumption is complicated in modern processors, depending on the voltage steps. To accurately come up with the configuration, we explore all possible frequency steps, and find the best configuration with equations (5.11) and (5.13). Because of the small number of possible frequency steps in a real system (usually between 10~20 distinct frequencies), the computation overhead of our approach is much less than 1 μs and is negligible.

Estimating Latency during Active Periods. To estimate the latency for requests arriving during busy periods, we use a similar method described above with no waiting time for the wake-up timer and no idle-active transitions. Although the request can arrive in the middle of a request processing, we conservatively assume that the time left for that request processing is the full request service time. We observed in our experiment that we lose less than 1% of power saving due to this assumption. The computation overhead is greatly reduced as we avoid dealing with conditional probabilities.

5.4.3 Minimizing State Transition Overheads

As shown in figure 5.11, transition overheads can account for significant portions of power consumption. In μDPM, we incur low power state transition overhead whenever a critical request triggers configuration updates. While effective at meeting the tail latency target, these configuration updates result in wasted power. It is because the core will sleep less and do no useful work during the state transition. In this section, we propose a *criticality-aware configuration scheme* to redirect requests and avoid unnecessary configuration updates. In addition, this also helps absorb bursty request spikes. Algorithm 1 shows the pseudo-code for this scheme.

When a request arrives at a core, there are 4 possible states that the request could face based on its criticality and core's status: (1) the request is critical and the core is sleeping (*critical-sleeping*); (2) the request is critical and the core is active (*critical-active*); (3) the request is not critical and the core is sleeping (*noncritical-sleeping*) and (4) the request is not critical and the core is active (*noncritical-active*).

When a request is either noncritical-sleeping or noncritical-active, no configuration update is needed, thus no additional state transition will occur. A noncritical-sleeping core is a core that is sleeping and already scheduled to wakeup so that the request will meet tail latency target. When a request is critical-sleeping, the wake-up time and/or frequency need to be updated. No additional C-state or VFS transition is introduced because the core is sleeping and only the frequency after wakeup and/or the scheduled time of wakeup are changed.

Algorithm 1 Criticality-aware scheduling

```
1: $non\_critical\_cores = \phi, non\_critical\_sleep\_cores = \phi$
2: for each core do
3:     compute $core_i$'s criticality
4:     if $criticality \leq 1$ then
5:         $non\_critical\_cores \leftarrow non\_critical\_cores \cup core_i$
6:         if $core_i$ is sleeping then
7:             $non\_critical\_sleep\_cores \leftarrow non\_critical\_sleep\_cores \cup core_i$
8:         end if
9:     end if
10: end for
11: if $non\_critical\_cores \neq \phi$ then
12:     if $non\_critical\_sleep\_cores \neq \phi$ then return $min\,(extra\ energy)$ in $non\_critical\_sleep\_cores$
13:     else return $min\,(extra\ energy)$ in $non\_critical\_cores$
14:     end if
15: else return $min\,(extra\ energy)$ in $all\ cores$
16: end if
```

The only case where a new transition is introduced is when the core is sleeping with no scheduled wakeup (the core is sleeping and have no arrived requests), and when a request is critical-active where an additional VFS transition is needed. The goal of the criticality-aware scheduler is to redirect the request to the core which makes it non-critical. By doing so, we decrease the chances of triggering configuration updates and VFS state changes.

Based on equation (5.12), we define the criticality score of a request for a core as:

$$criticality = \frac{S_f^{tail}}{A - A_{prev}} \tag{5.14}$$

, hence non-critical request will have criticality score less than 1. Our algorithm first computes the criticality score of that request for each core, and if there exist only one core which makes the request non-critical, that core is selected for processing. However, it is more than likely that multiple cores can make the incoming request non-critical. In this case, the non-critical sleeping cores are chosen for the request to avoid the possibilities for future VFS transition. If still multiple cores are candidates, we select the core which will process a request with the lowest *extra energy*. The extra energy for $core_i$ can be computed by:

$$E\left(W^i_{new}, f^i_{new}\right) - E\left(W^i_{cur}, f^i_{cur}\right) - S^{avg}_{f_{cur}} \times P^i_{f_{cur}} \tag{5.15}$$

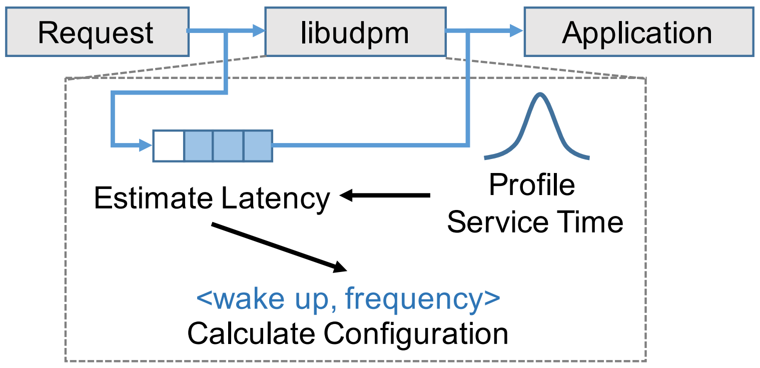
, where $E(W, f)$ is defined in equation (5.13), and (W_{new}, f_{new}) are the updated configuration as described in the previous section. For non-critical requests, no configuration update is involved ($f_{new} = f_{cur}$, $W_{new} = W_{cur}$), and this equation simply computes the energy to process a request under the core's current frequency configuration.

On the other hand, there are some cases that no core can make the request non-critical. In this situation, we follow the same least extra energy principle to redirect the request. Notice that we do not choose the core based on their operating frequency, since configuration update may involve both wake-up and frequency transition. By considering energy differences, we capture both transition overhead and the processing energy simultaneously, resulting in better performance and energy.

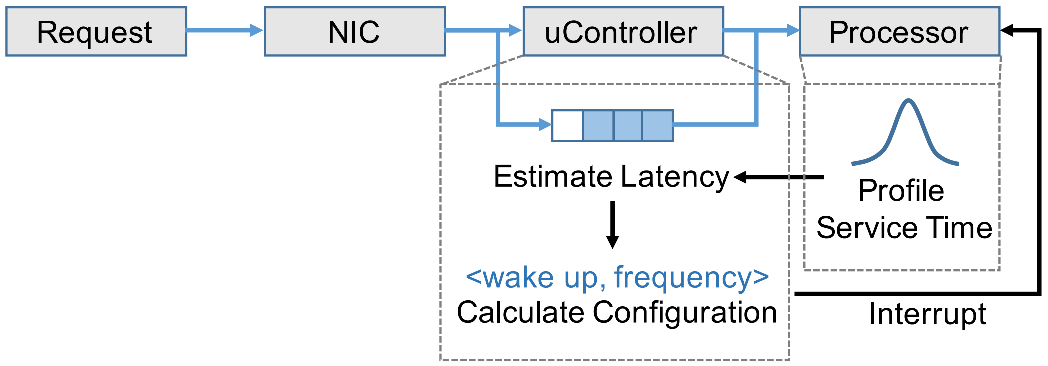
Another benefit brought from our criticality-aware scheduler is to mitigate bursty requests. When a burst of requests arrives, especially during prolonged sleep periods, most of the requests will be identified as critical. Our scheduler will attempt to schedule them to cores that make them "non-critical", resolving burstiness. From algorithm 1, we can see that the overhead of the criticality-aware scheduler is linear to the number of cores in the processor. In our experiment, this overhead is less than 2 μs for our 12-core Xeon processor. This overhead is considered as part of the request service time when deriving the configurations.

5.4.4 μDPM Implementation

Figure 5.15 shows both a software and hardware implementation option for μDPM. As shown previously in the section, the main components of μDPM is to profile the request service time, estimate the latency of incoming requests, and to calculate the new configuration for the processor. In a software implementation, μDPM can be implemented as a library (*libudpm*) and built on top of existing request handling libraries. An example of this



(a) Software Implementation



(b) Hardware Implementation

Figure 5.15: Implementation options of μDPM.

is libevent, an asynchronous event library, which memcached uses to receive requests. In a software μDPM, the library can profile the service time of each individual requests, and also estimate the latency of each incoming requests. If a request is identified as critical, we can have libudpm trigger a new configuration by modifying the processor frequency through existing cpufreq interfaces. To keep a core in a deep sleep state, libudpm will queue requests until the configurations wake up time. We prototyped the software libudpm implementation by modifying the libevent library used in memcached. In our 12-core evaluation server, we designated a single core for libudpm's thread in order to allow all other cores to be able to enter deep sleep. Using our prototype, we measured that the latency estimation model and configuration calculation incurs only a negligable 2μs overhead per request. Due to the lightweight overhead, we can further offload this to a dedicated hardware.

μDPM can also be implemented in hardware as a light-weight dedicated microcontroller which queues up requests and calculates configuration, with the assistance of software-side service time profiling. Only a single hardware controller is required per server. This configuration is similar to the Dream Processor in [93] and other low power embedded offloading cores [13, 15, 117]. When the hardware μDPM determines a configuration is required, an interrupt is triggered to the processor to modify the frequency [14].

5.4.5 μDPM Evaluation

In this section, we present our evaluation of μDPM. We evaluate μDPM using an in-house simulator with various latency-critical data center workloads representative of common microservices. Our simulator is a framework for stochastic discrete-time simulation of a generalized system driven by empirical profiles of a target workload (similar to BigHouse

[94]). Empirical inter-arrival and service distributions are collected from measurements of real systems at fine time-granularity. Using these distributions, synthetic arrival/service traces are generated and fed to a discrete-event simulation that models the server's active and idle low-power modes. Latency measures (e.g., tail response latency) are obtained by logging the start and finish time of each request. Similarly, energy measures are obtained through the weighted sum of the duration of idle, busy and transition periods with their corresponding power consumption. Similar to previous papers [33,46,68,69,86], we focus on CPU power as CPUs are the single largest contributor to server power. It was also observed that the CPUs' and server's power curve are of the same shape, therefore, CPU power is a strong proxy for server power [46,68].

The power consumption at each processor C-state and frequency step is collected from measurements of real systems and is shown in Table 5.1. Also, the uncore power (LLC and some peripheral circuit) is measured as 10W. We model a 12-core server, similar to our experimental server. Unless otherwise stated, we use 89 μs and 10 μs as sleep and VFS transition time, respectively. The sensitivity study of the state transition time is also given in section 5.4.5.

We validated our simulation infrastructure against our prototype as shown in Figure 5.16. The latency figure, which plots the tail latency of the baseline machine configuration, and the power figure clearly demonstrates that our simulation model agrees well with the actual behavior of the prototype system.

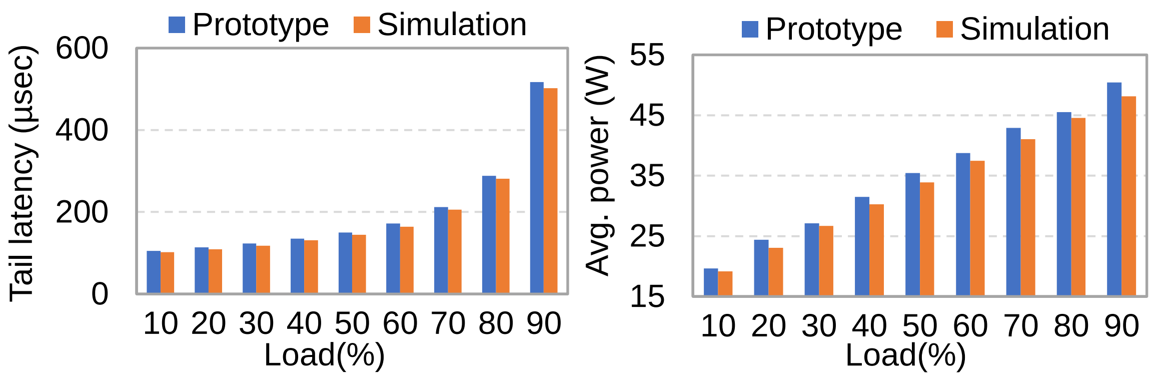


Figure 5.16: Prototype vs simulation validation.

Workloads

Typical web application may consist of front end web server (e.g. Apache, NGINX), business logic (e.g. accounts, catalog, inventory, ordering), databases or data stores (e.g. Memcached, MongoDB, Redis), or specialized functionalities, such as search engines and recommendation systems. In our evaluation, we select four different latency-critical workloads that are representative of typical backend microservices. Table 5.2 gives the workload characteristics of four applications we use in our evaluation. Throughout this paper, we define the tail latency at the 95th percentile and the target tail latency (SLA) as the tail latency of the applications running at medium load without applying any power management (similar to [69]). The service time was discussed in depth in section 5.3. We

Table 5.2: Workload characteristics.

Name	Avg. Service Time	Tail Service Time	Target Tail Latency
Memcached [7,47]	30μs	33μs	150μs
SPECjbb [69,70]	65μs	78μs	800μs
Masstree [69,70,90]	246μs	250μs	1100μs
Xapian [69,70]	431μs	1200μs	2100μs

later show the result of varying target tail latency and the effect on power savigns in section 5.4.5.

Memcached and **Masstree**: We evaluate a common in-memory key-value store used in many production data centers, such as Google, Facebook and Twitter. Due to the simple computational requirements, key-value stores involve significantly less processing time per request (tens of microseconds). Because of this, key-value store has even tighter tail latency constraints, typically on the order of hundreds of microseconds. Key-value store adds to our analysis an example of an ultra-low latency workload. We captured traces (request arrivals and service time) from a real server running *Memcached*. In addition, we use a different service time distribution from *Masstree* [90] provided in [69, 70].

SPECjbb: Business logic makes up a large component of many large-scale web applications, such as e-commerce and businesses. These business logic can include various functionality such as sales, inventory, and customer management. We evaluate with SPECjbb [10], a Java real-time middleware benchmark that simulates supermarket company with various microservices covering the services of supermarkets and suppliers (inventory, sales, orders, etc.), and headquarters (receipts, customer, etc.).

Search: We evaluate the query serving portion of a production web search service. Search requires thousands of leaf nodes all running in parallel in order to meet the stringent tail latency constraints. In search, most of the processing is in the leaf nodes that mine through their respective portion of data. Consequently, leaf nodes account for the vast majority of nodes in a search cluster and are responsible for an overwhelming fraction of power consumed [21, 22, 36]. We use the service distributions of *Xapian*, provided in [69, 70] as a representative search algorithm.

Results

We compare our results with three state-of-the-art power management schemes: Rubik [69] (VFS-only scheme), DynSleep [33] (sleep-only schemes), and SleepScale [85] (sleep+VFS scheme). The results of Rubik are representative of other VFS-only schemes, such as Pegasus and TimeTrader [86,114]. The original Rubik design assumes that frequency boosting is available. We model an optimistic Rubik design that sets the maximum frequency as the nominal frequency. SleepScale combines both Sleep state and VFS selection. SleepScale requires long computation time for its simulation-based prediction scheme, and therefore calculates Sleep and VFS selection after every 1 second epoch [85]. For DynSleep we set the frequency at maximum, and always enters C6 deep sleep state upon encountering an idle period. DynSleep is representative of other Deep Sleep techniques, such as PowerNap [91] and DreamWeaver [93], but outperforms both due to its ability to harness per-core level idleness instead of socket-level (full-system) idleness. In addition, we also show the results for the "Optimum" scheme which combines DynSleep and Rubik with no state transition overhead. The core will enter the deepest sleep state when idle and wake up instantly as soon as the request arrives, similar to DynSleep. During request processing, VFS configuration is adjusted at each request arrival instance, similar to Rubik. The idealized optimum scheme serves as an upper bound of energy saving that can be achieved under all workloads.

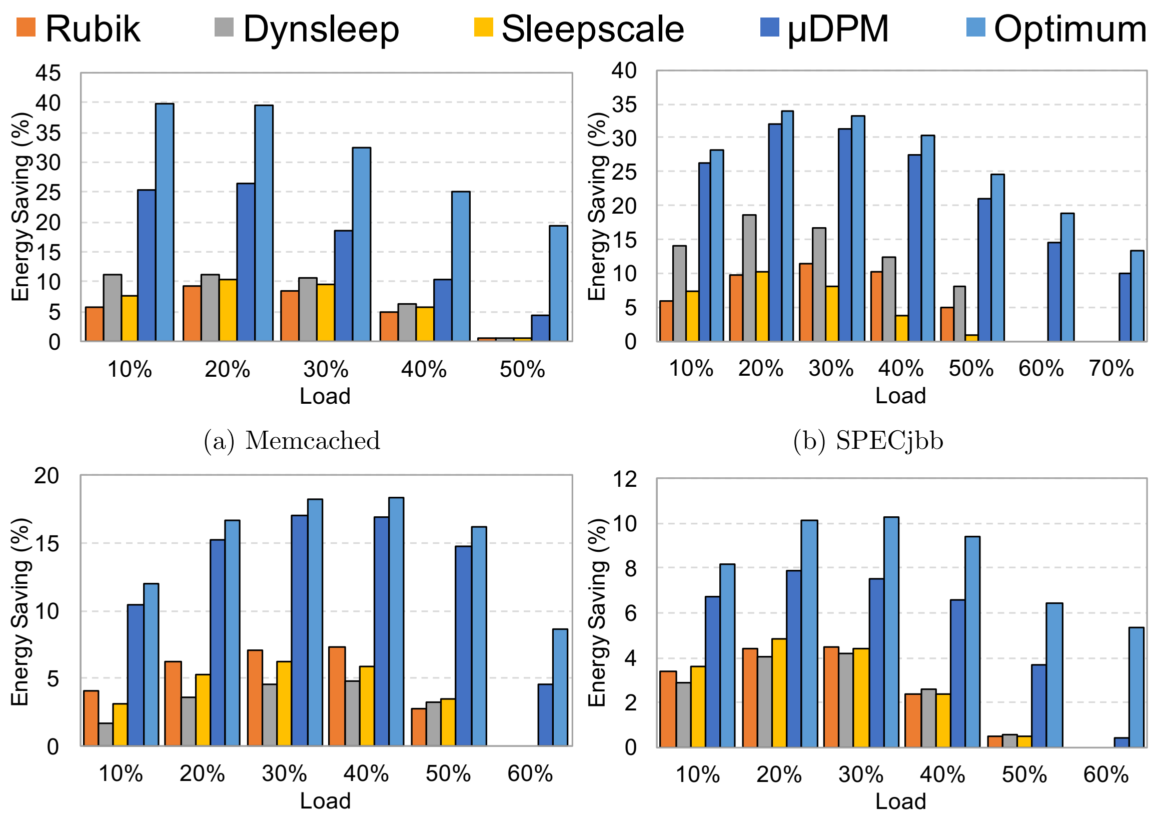
We consider processor level energy consumption, which includes both core and uncore power. For energy saving results, all schemes are compared with the default baseline scheme, where all CPU cores are operated under maximum frequency, and uses the Linux menu governor [103] to choose the C-state during the idle periods. We also use the Linux menu governor for C-state selection in Rubik, which only determine VFS configuration.

Energy Savings: In figure 5.11, we gave the energy consumption breakdowns and show that μDPM can effectively reduce busy and idle power, and the state transition power. μDPM only incurs a 3% energy overhead for transition. Here, we give the sensitivity studies of the energy saving on different traffic loads and applications. Figure 5.17 reports the energy saving results for four different applications across different CPU loads. The CPU loads are adjusted by scaling the inter-arrival time of requests.

Across all workloads, μDPM provides significantly more energy savings. Typically, μDPM is within 2-3% of the Optimum scheme, except for Memcached, which exhibits the shortest service times. Here, transition time overheads begin to dominate and μDPM starts to fall behind Optimum. Nevertheless, μDPM still provides ~2x energy savings compared to all other schemes.

Another noteworthy trend is that at high utilization, most schemes are not able to provide any power savings at all because the utilization exceeds the utilization that the target tail latency was set. However, in some cases, like in SPECjbb, Masstree and Xapian, μDPM is still able to squeeze out modest energy savings where all other DPM fails. These results demonstrate that request delaying and sleep states are the key to saving power at high utilization with sub-millisecond latency constraints.

For extremely low latency workloads (*Memcached* and *SPECjbb*), the major energy inefficiency is the idle period energy consumption as demonstrated earlier in this paper.



(a) Memcached (b) SPECjbb

(c) Masstree (d) Xapian

Figure 5.17: Energy saving comparisons among different power management schemes.

Therefore, sleep-based techniques provide better energy savings than VFS-based techniques. SleepScale achieves good energy saving by selecting the best C-state and VFS pair through design space exploration. However, because of the short natural idle periods, cores can only enter shallow sleep states during the short idle periods. Therefore, significant energy saving can be achieved by delaying request processing to create longer idle periods, as in the case of DynSleep. With μDPM, we can achieve up to 26% energy saving for *Memcached* and 32% for *SPECjbb*, compared with the next best technique, DynSleep, at 12% and 18%, respectively.

For applications with relatively higher request processing time (*Masstree* and

Xapian), the achieved energy savings are relatively lower as the baseline scheme is also able to take advantage of longer natural idle periods and enter deep sleep states. All existing schemes perform similar in terms of energy saving. DynSleep provides only moderate energy saving. Longer processing time means longer nature idleness, and further prolonging these idle periods will only give marginal return in energy saving. SleepScale typically outperforms all other schemes, except for μDPM, because it chooses the C-state and VFS configuration with lowest power by exploring the entire C-state and VFS's design space. Longer idle periods here allows SleepScale to naturally enter deeper sleep states. With μDPM, we achieve up to 17% energy saving for *Masstree* and 8% for *Xapian*, compared with the next best technique, at 8% and 5%, respectively.

Contrary to existing assumptions that sleep states cannot be used in latency-critical workloads [69,86,114], *these results demonstrate the need to coordinate delay request processing, sleep and VFS power management with microsecond latency constraints.*

Reducing State Transition Overhead: Figure 5.18 shows the normalized state transition count with our criticality-awareness algorithm. The results are normalized to the sum of C-state and VFS transition counts in μDPM without criticality-awareness scheduling. We can see that criticality-awareness effectively reduces the C-state and VFS transition overhead to less than 20% of the baseline μDPM. For all applications, at low load, the majority overhead reduction is for VFS, and at high load, for C-state. At low load, it is easier to find the sleeping core which makes requests non-critical. As a result, more VFS transition is avoided. On the other hand, it is harder to find cores that can make requests non-critical at high load, hence criticality-awareness will try to avoid C-state transition since it has

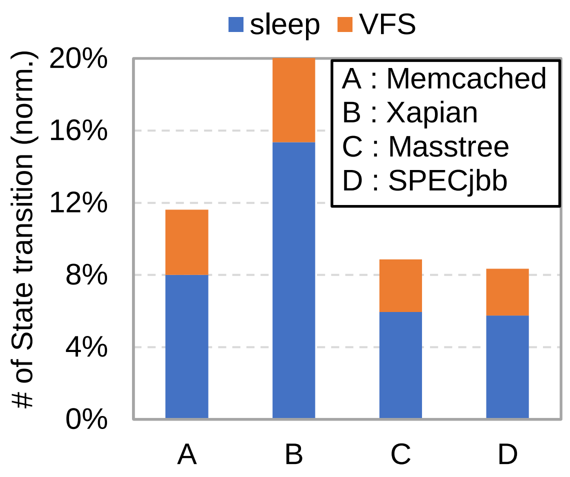


Figure 5.18: State transition reduction with criticality-awareness.

higher power overheads.

Responsiveness to Load Changes: In addition to meeting the target tail latency in the steady state, μDPM's design allows it to respond instantaneously to sudden changes in input load. Since the wake-up time and frequency are updated at every request arrival, μDPM reacts to the changes in the inter-request arrival time immediately (Rubik and DynSleep has similar behavior). Figure 5.19 shows how μDPM responds to sudden load change with *Masstree*. As a comparison, we also show the tail latency with another Sleep+VFS scheme, SleepScale, using a 60s epoch as in [85]. The input load trace is shown on the right, which we gather from our institution's data center. The target tail latency is defined as the tail latency under maximum load (80%) from the trace. The 95$^{\text{th}}$ percentile latency for SleepScale and μDPM, calculated every 1 second, is shown in the left. First, we can see that SleepScale can not close the latency gap under low load. As discussed previously, SleepScale close the latency slack by VFS, which provides limited room for slowing

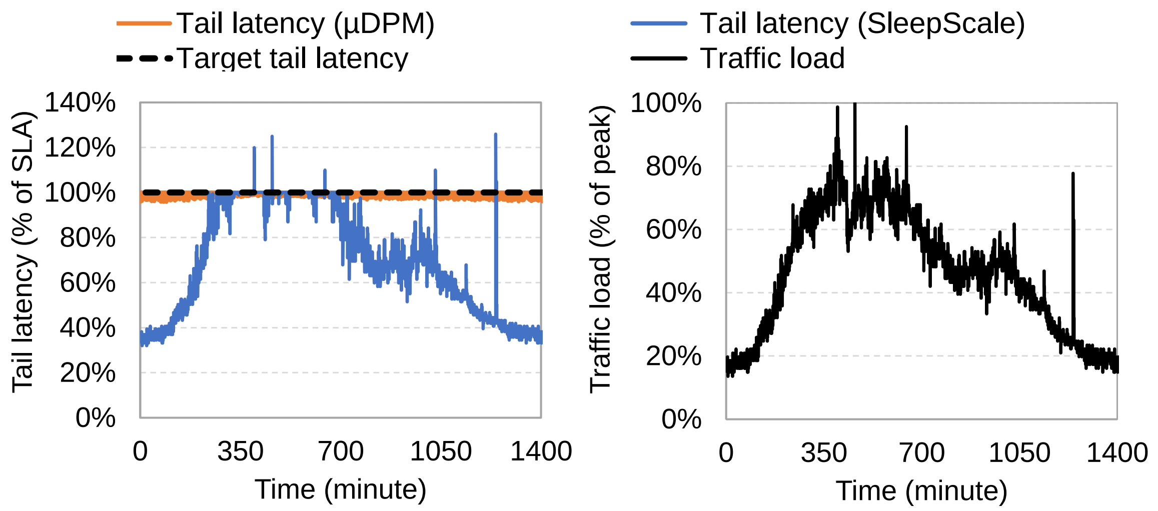
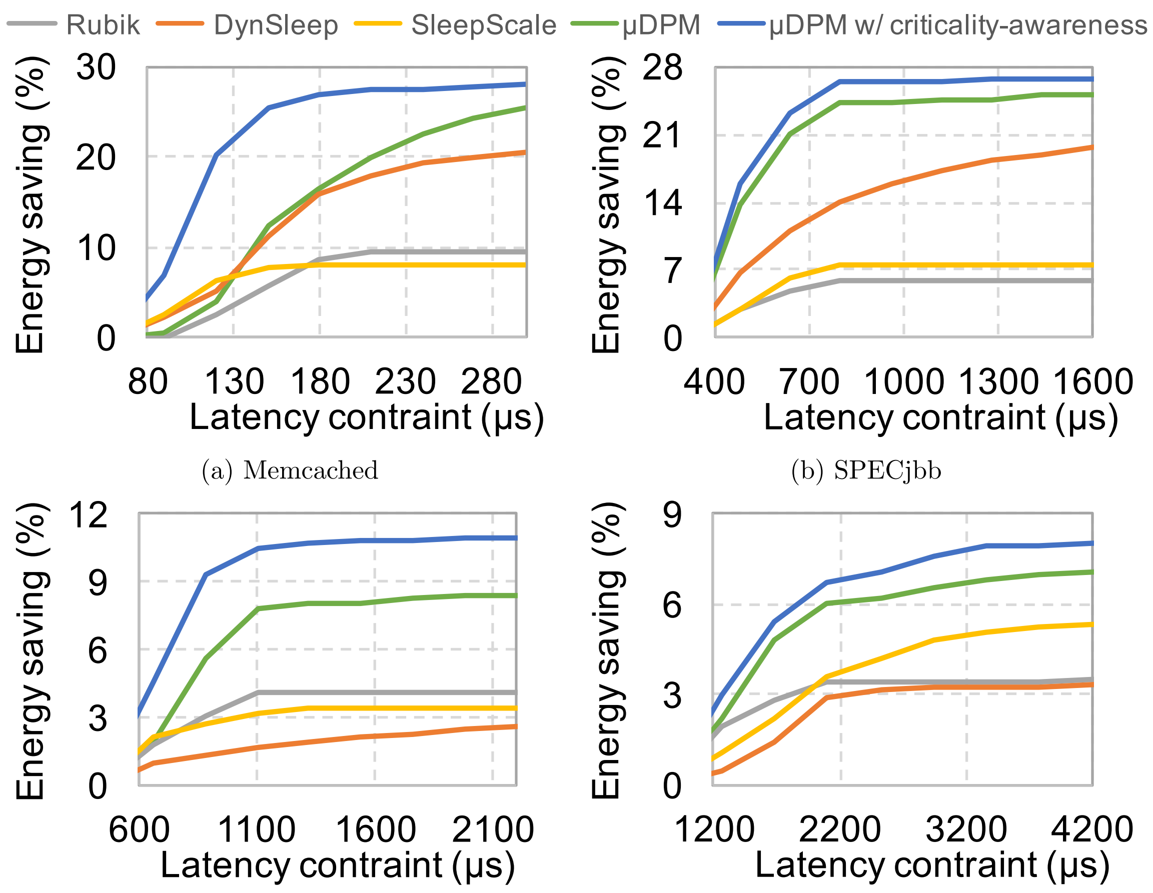


Figure 5.19: Tail latency under varying traffic load.

down request processing. Also, SleepScale can not always satisfy the target tail latency; due to its history-based prediction, it can not react to sudden load changes, resulting in multiple target tail latency violations. On the contrary, we can see that μDPM is able to achieve stable tail latencies under all loads. At low load, μDPM chooses to delay requests and wake up the core later, process requests at a lower rate, and achieve tail latency close to the latency bound while saving additional power. Also, it adapts quickly to sudden load changes and burst, thanks to the dynamic wake-up time and frequency adjustments.

Sensitivity to Tail Latency Constraint: In data centers, the target tail latency is usually defined at the peak load. Previously, we assumed that the SLA is set at 50% load. However, there might be some cases that the peak load is different.

Figure 5.20 shows the energy savings with different tail latency constraints. The x-axis is the latency constraint for each application, and the y-axis is the energy savings achieved for that target tail latency constraint. As expected, energy savings generally de-



(a) Memcached (b) SPECjbb (c) Masstree (d) Xapian

Figure 5.20: Sensitivity to target tail latency.

creases as the SLA gets tighter in all four applications and all schemes. The rate of this decrease is highly related to the application characteristics. For Memcached, the energy saving quickly drops to 0 as latency constraint decreases under all schemes, except μDPM with criticality-awareness because the latency slacks become smaller than the resident time of deep sleep state and the VFS transition latency. We achieve better energy saving as latency constraint decreases due to the coordination request delaying and request rescheduling among cores. For other workloads, the energy savings are more resistant to the latency constraint because of the relatively larger latency slack. μDPM consistently outperforms all

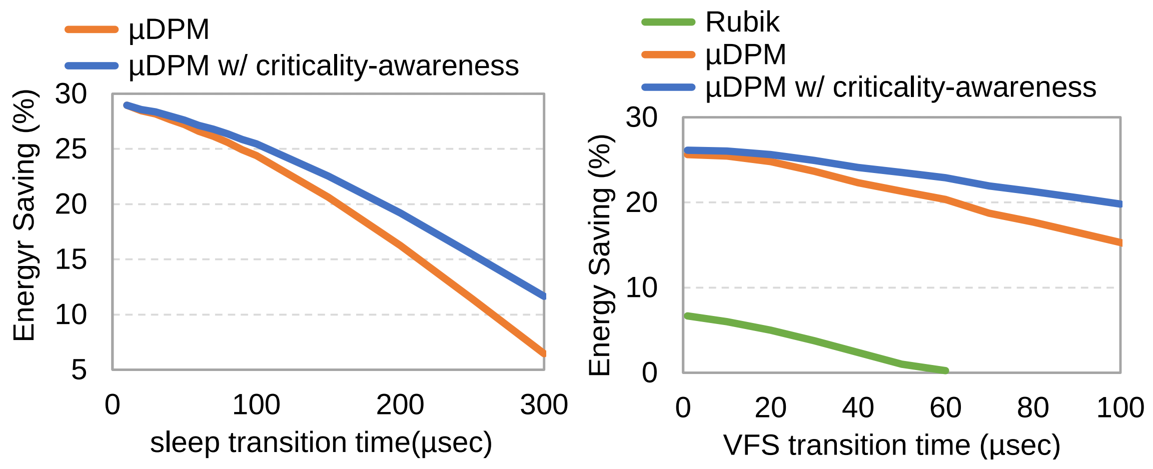


Figure 5.21: Sensitivity to transition time.

other schemes in four applications and all latency constraint levels. *µDPM consistently outperforms all other schemes by saving power over 3x in Masstree, and 2x in SPECjbb and Xapian.*

Sensitivity to State Transition Overhead: To understand the utility of µDPM, we characterize its effectiveness by varying sleep or VFS transition times. Figure 5.21 illustrates how energy saving decreases with increasing transition time. We present results for *Memcached* at 10% load. Since µDPM determines the configuration considering both sleep and VFS transition overheads, it will dynamically balance the use of each scheme. As the sleep transition increases, the energy saving from sleep decreases, and most of the energy saving are from VFS. On the other hand, as the VFS transition time increases, µDPM will tend to use more sleep to achieve energy saving. Also, we present the results for Rubik, in the right hand side of Figure 5.21. Clearly, µDPM is more resistant to the VFS transition overhead. On top of that, with criticality-awareness, the energy saving is more resistive to the increase of transition overhead because of the state change reductions.

Chapter 6

Power Management under Quality and Latency constraint

6.1 Introduction

Web search is an important class of data-intensive online services in data centers, serving billions of users each day. Needless to say, search engines need to meet strict performance constraints, which directly affect users' satisfaction as well as search engine revenues [108]. Nonetheless, this is a very challenging task: search engines need to process tens of thousands of queries every second over billions of documents, but users expect the response time to be under a few hundred milliseconds. Thus, although high-quality content closely relevant to users' queries are crucial in search results, search engines may often have to compromise the search quality in order to meet the stringent latency requirement. Additionally, power consumption of a search system is also a critical performance metric.

With hundreds of thousands of machines, significant power consumption is already leading to a soaring operating cost for the search service provider. As a result, reducing power consumption while meeting the quality and latency constraint is a major challenge in designing a distributed search system.

Query processing in search engines often involves multiple retrieval stages and a distributed architecture. Specifically, upon receiving a query, the aggregator dispatches it to the ISNs and each ISN starts retrieving top-K ranked results for the given query on its own web index database. After that, the top-K results from each ISN are collected by the aggregator and the results are re-ranked to find the final top-K results. Since the aggregator needs to wait for all responses from the ISNs, the response time depends on the slowest ISN. Due to significant response time variation among the ISNs, a long latency at any ISN manifests as a slow query response at the aggregator level [18, 37, 71]. At each ISN, returning top-K results requires exhaustively computing the relevance score for each indexed document, which is time consuming and often unnecessary since most of the evaluated documents will not make to the ISN-level top-K results. Moreover, most of the results at ISNs are not used because only the top-K results from the aggregator will be sent back to the user. Clearly, these unnecessary computations, albeit useful for producing high-quality search results, lead to long query response time and power wastage.

Several techniques have been proposed to reduce request processing time at the ISN level. For example, dynamic pruning (or early termination) reduces the number of documents to be evaluated by skipping the documents that can not make to the top-K results [39, 40]. Although the pruning methods reduce some unnecessary computations and improve the average query processing time at the ISN, the query processing time variation among ISNs is still high [65]. As a result, the improvement of the query response time at the aggregator level is limited. Another method is proposed to employ an aggregator timeout, which determines how long the aggregator waits for its ISNs before sending the results back to users [120]. The key idea is to trade off the result quality and the response time: the shorter the aggregator waits, the less ISN response it collects, which improves responsiveness but potentially degrades the quality due to possible missing of results from certain ISNs. Nonetheless, although the aggregator policies improve the response time subject to a quality constraint, they did not address the power consumption issue, as the ISNs will still perform the query processing for all queries and result in unnecessary power wastage. Further, [120] defines the quality as the number of responses received from the ISNs, but this does not capture the characteristic of a distributed search engine that different ISN responses may contribute to the final result differently due to ISN heterogeneity.

Some prior studies have proposed to save the data center server power by slowing down or postponing the query processing through voltage and frequency scaling (VFS) and CPU sleep states while meeting the tail latency constraint [33,69,85,86,93]. However, these studies do not take into account specific workload characteristics in search systems. For example, our experiments show that there exists a correlation among ISN query processing times in the search system, whose impact on quality has not been well studied to improve energy efficiency. Finally, the previous work [33, 69, 85, 86, 93] considers a single server, not a multilevel search system with aggregators and ISNs in a realistic search system. As a result, they do not provide good system-wide energy efficiency improvement while meeting the aggregator-level tail latency and response quality constraints.

In this chapter, our goal is to develop a cooperative aggregator-ISN policy that cuts out the slow ISN responses for fast response time and reduces energy consumption. First, at the aggregator, a "time budget" is assigned to the ISNs for request processing based on the expected quality level. Our quality definition is based on a realistic approach which compares returned documents in the response to the documents in the "golden" benchmark results of the query when it is processed completely. Based on the budget, each ISN serves the request only if the predicted processing time is within the time budget, and disregards the request for energy saving otherwise. Furthermore, the aggregator may choose to re-send the query for full execution in case the response quality is not satisfactory. By doing so, we essentially reduce the ISN processing workload and aggregator level latency, while still meeting the quality constraint. Since the time budget is known to the ISNs, we propose to proactively put selected ISNs to sleep state to save power when the time budget cannot be met. Our experimental results show that TailCut can achieve up to 25% and 14% reduction on latency and power consumption, respectively, while satisfying the desired 99% quality constraint.

To summarize, we make the following contributions:

- We experimentally verify the query processing time and variation among the ISNs, which greatly impacts not only the aggregator-level query response time but also the energy consumption of ISNs.
- We define a realistic ISN level response quality, and illustrate through extensive experimental results that ISN queries that need longer processing time do not necessarily

give better ISN response quality.

- We propose TailCut, a cooperative aggregator-ISN policy, which intelligently discards /processes the query processing at the ISN level. We also design an on-line algorithm which monitors and dynamically adjusts the TailCut design parameters to minimize power consumption while maintaining the response quality and tail latency under pre-defined constraints.
- We evaluate the effectiveness of TailCut using an in-house simulator for large scale distributed search system. We gather query traces by deploying Lucene search application with Wikipedia datasets on machines, and feed into our simulator.

6.2 Search Engines Overview

Typically, search engines employ multiple retrieval stages and a distributed architecture with hundreds of thousands of machines. Figure 6.1 illustrates a common architecture of an index serving system consisting of index serving nodes (ISNs) and a single or multiple levels of aggregators. An entire document index, containing information on billions of documents, is document-sharded [21] and distributed among a large number of ISNs. To balance the number of indexes among ISNs, each document is first hashed based on the content of the document. Each ISN is assigned with a range of hash values and the document indexes are then distributed based on its hashed value.

When a user sends a query and the results for its query are not cached in the frontend, the aggregator will propagate the query to all ISNs hosting the web document indexes. Each ISN searches its fragment of the index to retrieve the local top-K matching

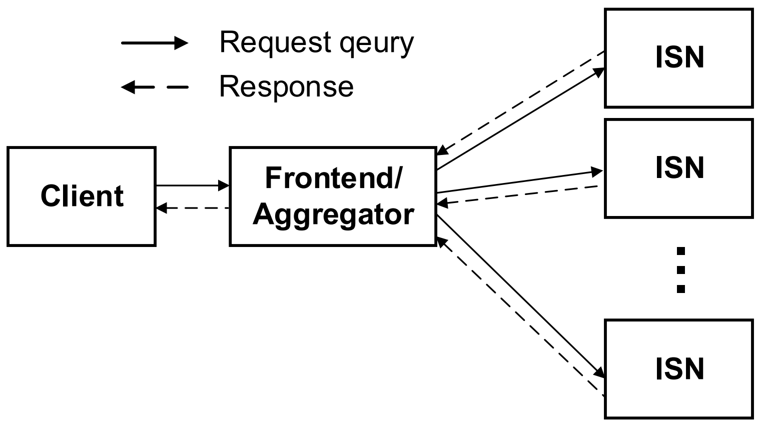


Figure 6.1: Search engine architecture.

documents and sends them to the aggregator. Then, the aggregator merges them, and computes the overall top-K results among all the returned ISN results.

The ISNs are the most important part of the index serving system and account for the majority of the total end-to-end latency and most of the power consumption in the search system is caused by the ISNs [65]. Upon receiving a query, each ISN will search its index shard to return the matching documents of a query. The documents inside the index shard are sorted based on static scores (such as PageRank [25]) reflecting the popularity and importance of the document. When an ISN determines that a document matches the user query, it computes the relevance score using a ranking function that combines many factors including the document static score, term features (such as term frequency in the document [104]) and other features (such as proximity to user location or history). The query processing follows “document-at-a-time (DAAT)” [26]. That is, the score of a document is fully computed before moving to the next document in the postings list. For a multi-term query, the intersection/union of the postings list of each term will be traversed to find the top scores documents.

6.3 Problem Formulation

A distributed search system is typically evaluated in terms of three performance metrics: query response latency, answer/search quality, and power consumption. Next, we formalize the design of TailCut.

Latency. User queries are often non-deterministic in term of the query context and arrival time, hence the time to answer the query cannot be known beforehand. We define user query latency as the time measured between the aggregator receiving a user query and sending the reply back to the user. Search system often requires not only low latency, but also low latency variance. Thus, operators typically resort to maintain a high-percentile latency of user queries, also called tail latency, under a satisfactory level to ensure that most users receive timely responses. In this chapter, we use the widely-adopted 95%ile latency as a concrete example, although higher-percentile latency can also be considered.

Quality. To quantify answer/search quality, we generalize [55] and define the quality as:

$$Q = \frac{\sum_{n \in R_{user}} S_n}{\sum_{k \in R_{golden}} S_k} \times 100\% \tag{6.1}$$

which is the ratio of the total score of the matched documents sent to the user, R_{user}, to that the system would otherwise get without any latency constraint, R_{golden}. The prior work [120] defines the quality as the ratio of number of responses received from ISNs by a time over the number of responses without any time limit. However, this is less accurate in top-K processing system, since it ignores the ISN heterogeneity in search quality contribution. For example, if the responses from certain ISNs do not contain any documents that will be included in the final answer sent back to the user, missing responses from those ISN will not affect the final answer quality.

Power consumption. A CPU core runs at a full power when processing a query, and becomes idle and enters a deep sleep state with little power consumption otherwise. Let the number of ISNs in the system be N, and the number of queries arrived during T_{tot} be U. Then, the average system power consumption can be modeled as:

$$P_{sys}^{N,U} = N \times P_{peri} + \sum_{n=1}^{N} \frac{\sum_{q=1}^{U} P_{proc} \times T_{q,n}}{T_{tot}} \tag{6.2}$$

where $T_{q,n}$ is the time spent on processing query q on ISN n and P_{peri} is the non-CPU power consumption of an ISN. Since the server power consumption is dominated by the CPUs [68], we will focus on the CPU power consumption.

Optimization objective Our objective is to minimize energy consumption of a distributed search system with N ISNs, while satisfying tail latency and quality constraint constraints. Given a set of U queries, we denote the latency and quality of query q by L_q and Q_q, respectively, for $q = 1, 2, \ldots, U$. The tail (95%ile) latency for the set of U queries, denoted by L_{tail}, is the $0.95 \times U$-th largest latency among these U queries. Thus, we formulate the power minimization problem as follows:

$$\text{minimize}_U \qquad P_{sys}^{N} \tag{6.3}$$

$$s.t., \qquad L_{tail} \leq \overline{L}, \text{ and } \frac{1}{U} \sum_{q=1}^{U} Q_q \geq \overline{Q}. \tag{6.4}$$

where $\overline{L}$ and $\overline{Q}$ is the tail latency and average quality constraints, respectively.

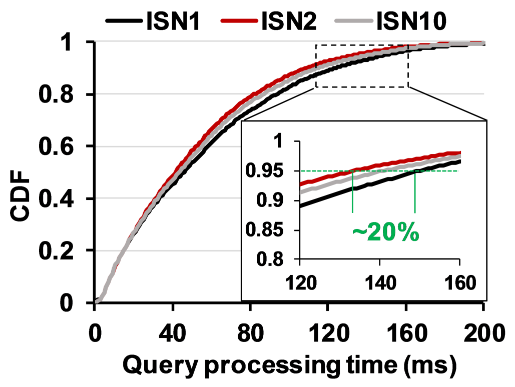
6.4 Search Engine Characteristics

Search system employing a distributed architecture exhibits several characteristics that are instrumental for TailCut. In this section, we conduct extensive experiments to reveal them, based on an experimental setup given in section 6.7.1. Our main findings are:

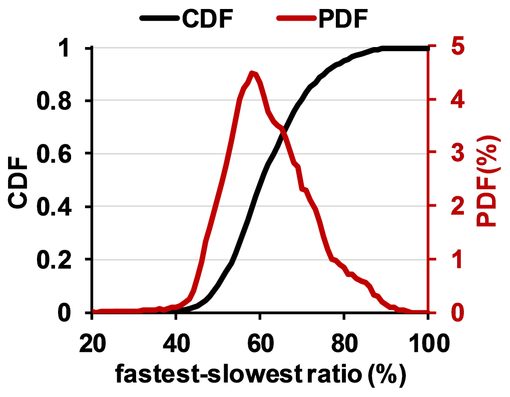
- There exists a high variance, both within and across ISNs in the query processing time.
- The query processing time is correlated among ISNs.
- Slow processing queries contribute significantly to the total energy consumption.
- The ISN query with long processing time does not produce a high contribution to the final result.

6.4.1 Processing time

Based on the partition-aggregate model and high processing time variance among ISNs, *the aggregator-level response time is determined by the slowest responding ISN*. Although the aggregator propagates the same query to all ISNs, ISNs will need to traverse different postings lists hence spend different time to process these queries. Figure 6.2(a) shows experimental results of the query processing time distribution of ISNs. For the simplicity of presentation, we only show the CDF of the query processing time from 3 ISNs. First, we can see that the processing times vary significantly from query to query. The maximum processing time can be 5X of the average processing time. Further, contrary to the previous assumption that the CDF is the same across all ISNs [69, 86, 120], we see that processing time is not uniform across ISNs. For example, ISN10 spends 10% more time on query processing than ISN1 on average. For the high percentile processing time (95%ile



(a)



(b)

Figure 6.2: ISNs processing times diversity. (a) Query processing time distribution within ISNs. (b) The fastest-slowest ratio of the query processing time across ISNs.

in figure 6.2(a)), the difference is even more significant (~20%). This difference is due to heterogeneous index distributions among ISNs. For each term, its postings list may not have equal lengths among ISNs, resulting in different query processing time distributions at different ISNs [71].

To further show the discrepancies of query processing time among ISNs, we conduct query level analysis. Figure 6.2(b) shows the distribution of the *fastest-slowest ratio* (FSR) of the queries. FSR is defined as the ratio between the query processing time of the fastest and slowest ISN for the same query propagated from the aggregator. The lower the FSR, the higher the discrepancies among ISNs. In figure 6.2(b), we can see that about 50% of the queries have a FSR less than 60%. Also, more than 90% of the queries a have FSR between 40 and 80%. Clearly, there exists a significant query processing time variation among ISNs.

These results lead to our first observation:

Observation 1 *There exists a high variance, both within each ISN and across ISNs in the query processing time.*

To further analyze this variation, we also present the processing time distribution among ISNs in figure 6.3. Different from figure 6.2(a) that gives the processing time distribution for each ISN, figure 6.3 focuses on the "differences" between ISNs. First, we find the query processing time α of the first ISN, and log the *exceeding ratio* of all ISNs and queries. The exceeding ratio is defined as how many ISNs have query processing time larger than certain time (β), normalized to the total number of ISNs. For example, a point (50,0.6) on the left most curve means that if the first ISN has query processing time between 50 and 60 ms, 60% of ISNs will have the query processing time larger than 50 ms on average. In this figure, we can see that there exists a *correlation* among query processing time of ISNs. With different α, the query processing time distribution among ISNs is also different. For example, with $50 \leq \alpha < 60$ms, no ISN in the system will spend more than 100 ms for the same query propagated from the aggregator. On the other hand, if $100 \leq \alpha < 110$ms, about 60% of ISNs will spent more than 100 ms. The results indicate that a long/short request

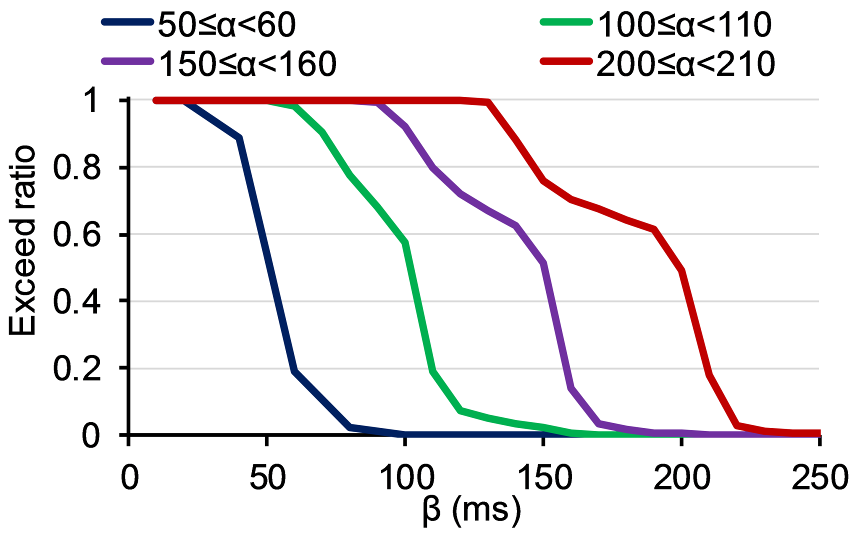


Figure 6.3: Correlation among ISNs' query processing time.

on one ISN is also likely to be long/short on other ISNs. Such correlation is due to the ISN index balancing when adding new indexes to ISNs. Hashing the document can effectively balance the number of indexes among ISNs. But imbalance among per-term postings list still exists since the per-term postings list is a subset of the ISN indexes. As a result, the postings lists among ISNs for the same query terms are imbalanced yet correlated. Note that we choose the first ISN as the base of our analysis only for convenience. Choosing other ISNs will yield similar results.

Observation 2 *There exists a correlation among ISNs' query processing time.*

6.4.2 Energy consumption

Since search engine is based on a top-K query processing model, most of the top-K documents returned by ISNs are not used for the final results (aggregator-level top-K results). For example, an a search system with 1000 ISNs, in total 1000K documents are returned by ISNs, but only K documents are needed and sent back user response. Hence, more than 99% of ISN computation results are discarded, which leads to a high energy

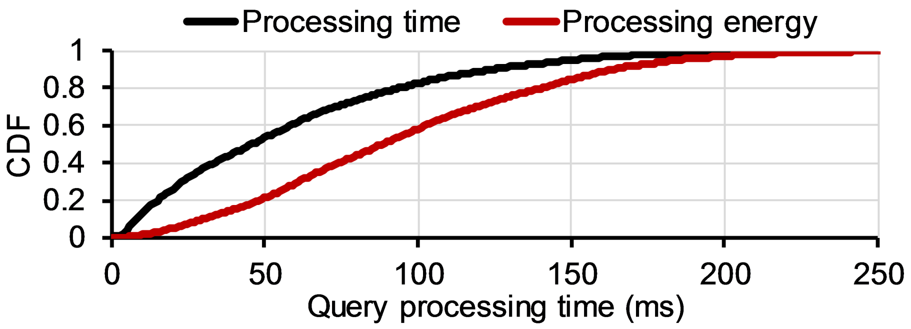


Figure 6.4: Query processing time and energy distribution.

wastage. In addition, we also analyze the energy consumption behavior at the ISN level. Figure 6.4 shows the query-wide statistics (CDF) of the processing time and energy. The CDF of processing energy should be interpreted as: the portion of energy that is spent on queries with processing time less or equal to a certain processing time. From this figure, we can see that ~50% of queries have processing time less than 50ms, and these queries only consume ~20% of total energy; the slowest 20% of queries consume 40% of energy; the slowest 5% of queries comsumes ~20% of energy. The long processing time queries clearly dominates the energy consumption.

Observation 3 *A small portion of slow processing queries contribute significantly to the total energy consumption.*

6.4.3 Query response quality

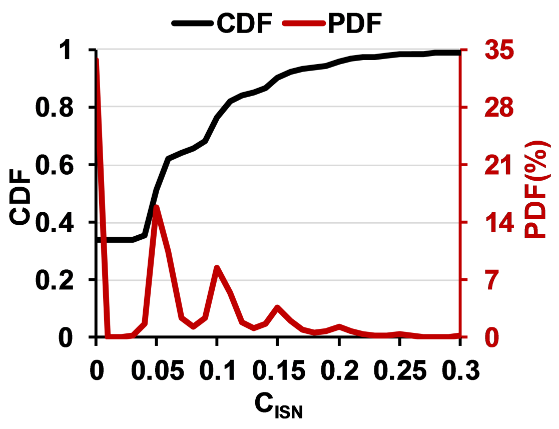
Besides query processing time and energy, the search result quality (i.e., how relevant the returned results are to the user's query) is another important metric for a search engine. Previous works have shown that the quality of search engines exhibit an "adaptive" nature at both ISN and aggregator level [55, 120]. At the ISN level, processing a request for more time improves the ISN-level response quality. This adaptive execution may return lower-quality responses (or partial results) for responsiveness [56, 62, 105]. On the other hand, at the aggregator level, an aggregation policy determines how long the aggregaot waits for the ISNs before sending the aggregated top-K results back to users. The longer the aggregator waits, the more results it collects, thus improving quality but degrading responsiveness [120]. While prior studies have exploited the tradeoff between query response time and quality to reduce tail latency, they only focus on a single level, either ISN or aggregator. The tradeoff between ISN-level processing time and the aggregator-level quality remains under-explored. Further, the prior work [120] defines the quality as the ratio of number of responses received from ISNs over total request sent to ISNs. However, this does not capture the fact that different ISN responses contribute to the final result differently (and that most ISN results are discarded by the aggregator).

In this chapter, we define the quality contribution of an ISN, C_{ISN}, as:

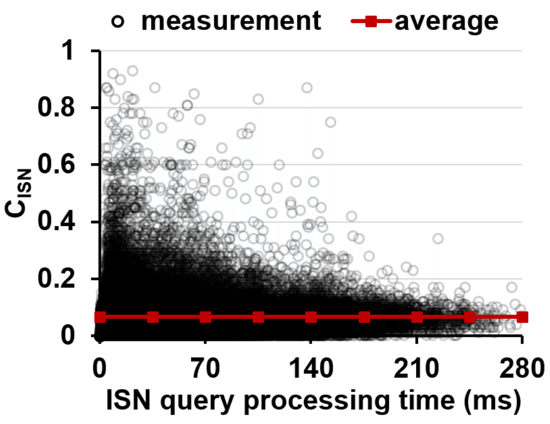
$$C_{ISN} = \frac{\sum_{n \in \{R_{ISN} \cap R_{AGG}\}} S(n)}{\sum_{k \in R_{AGG}} S(k)} \tag{6.5}$$

where R_{ISN} and R_{AGG} are the top-K results at the ISN and aggregator, respectively, $S(n)$ is the score of the document n in top-K results. This quality definition is variant of the weighted quality [55], and reflects the different contributions of different ISNs to the overall search quality. In particular, $C_{ISN} = 1$ means that the result from a particular ISN is exactly the result sent back to user by the aggregator, and $C_{ISN} = 0$ means that the result from this ISN is completely discarded by the aggregator and has no contribution to the final top-k results.

Figure 6.5 shows the experimental results on the ISN contribution. In these experiments, we use 16 ISNs with $K = 20$, i.e., each ISN returns its top-20 results and the aggregator compiles the final top-20 results from the ISNs and sends them back to the user. Figure 6.5(a) shows the query-wide ISN contribution distribution. We can see that more than 30% of the time the ISN contribution is 0, and about 80% of the time it is less than 0.1.



(a)



(b)

Figure 6.5: The ISN quality contribution. (a) query-wide distribution, and (b) relationship between ISN contribution and processing time.

This means that most of the time an ISN only contributes little or none to the final results sent back to users. In figure 6.5(b), we plot the per-query ISN contribution/processing time relationship, where x-axis is the query processing time and y-axis is the ISN contribution. The reading of each query are in black circles. The solid red line represents the average. We can clearly see that the query processing time has little correlation with the average ISN contribution. Specifically, a higher processing time does not mean that the ISN will contribute more to the final results. This also contrasts with the conventional wisdom that a higher processing time at the ISN improves the quality [56], and motivates our study. In addition, with a lower query processing time, the ISN quality *variation* is larger. This can be due to the unbalanced data partitioning among ISNs.

Observation 4 *The ISN contribution is low most of the time, and a longer ISN processing time does not necessarily produce a higher contribution.*

The above four important experimental observations motivate us to develop TailCut, a quality-aware scheme that can improve both responsiveness and energy consumption in a distributed search system.

6.5 TailCut Design

In section 6.4, we found that the long processing time of some ISN queries is the major factor limiting the search system performance. These ISN queries not only degrade the latency of user queries, but also lead to high ISN energy consumption. Furthermore, these ISN queries do not contribute significantly to the response quality. In this section, we will present TailCut, an aggregator-ISN cooperative policy that judiciously executes queries considering the processing time, energy and quality.

6.5.1 TailCut Overview

TailCut is a policy that determines the "time budget" for each ISN to process the request. Each ISN will predict the time needed to process an incoming query and compare it with the allocated time budget. The ISN will only execute the query when the predicted processing time is shorter than the time budget. By doing so, we essentially reduce the ISN processing workloads at the potential cost of slight quality degradation. However, by dropping a query, the ISN will go to sleep and save power, and the tail latency can be effectively reduced.

Figure 6.6 shows the overall system architecture of TailCut with one aggregator and multiple ISNs. When the front-end server receives a search query from the user, it

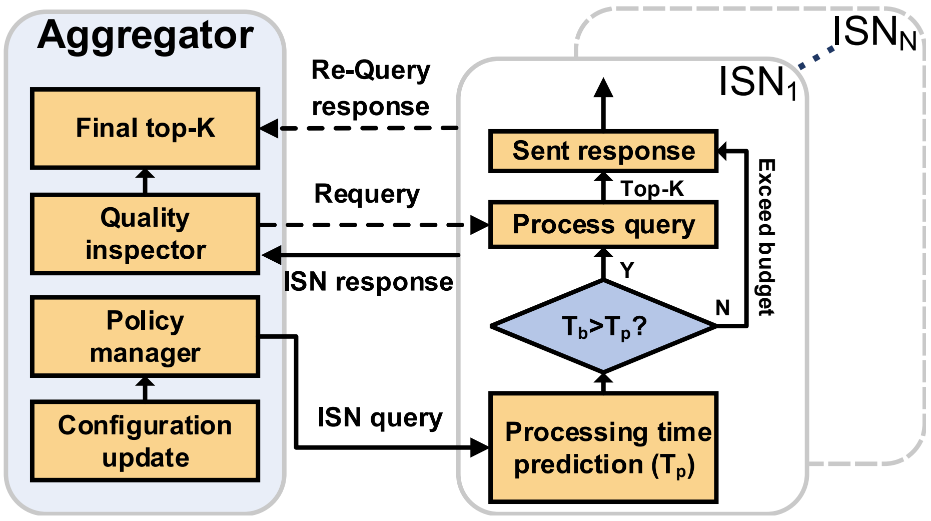


Figure 6.6: TailCut system overview.

forwards the query to the aggregator. The aggregator will then broadcast the query to every ISN along with the "ISN time budget (T_b)". This ISN time budget limits how long an ISN can spend on processing this query. At each ISN, the query will first be inserted into its FIFO queue, and an idle thread will fetch the query from the queue and process it when the CPU becomes available. Before traversing the postings lists, a thread will predict how long it will take to process the query, T_p, based on the summation of the length of each term's postings list. The regular query processing flow will be applied if the predicted processing time is shorter than the query time budget, $T_p < T_b$. Otherwise, the query processing is skipped and a response with "exceed budget" flag will be sent back to the aggregator.

The aggregator consists of four parts: policy manager, quality inspector, final top-K computation and configuration update. After broadcasting a search query, the aggregator will wait for the replies from the ISNs. The collected results will first go through the quality inspector. The quality inspector will wait for a period of time, T_w, and check how many ISNs respond with "exceed budget". The quality inspector can then know the number of

ISNs which drop their query processing. If this number is small, the inspector will wait for responses from the remaining ISNs which process their ISN queries, and send the collected responses to the final top-K computation. In this case, ISN results with long processing time are dropped, thus reducing the search query latency and ISNs energy consumption. On the other hand, if the number of dropping ISNs is large, it will lead to significant quality degradation, since only few ISNs will respond with their results. To prevent high quality degradation, the quality inspector will issue a full query execution, called "Requery", which re-sends the query to ISNs without any time budget. After all requery responses are received, if any, all ISN responses (from both the initial query and requery) will be sent to the final top-K computation to compile the final response to the user search query.

Clearly, the performance of TailCut is affected by the time budget, requery, and query characteristics. In general, frequent requeries adversely affects the response time, since the response time includes both the quality inspector waiting time (T_w) and the longest requery processing time among ISNs. To prevent frequent requeries, either ISN time budget needs to be set larger or the quality threshold that determines whether to send requeries needs to be set lower. On the other hand, having a large ISN time budget and/or a low quality threshold for requeries will increase latency and/or energy, since more queries will go through full execution. The configuration update will dynamically adjust both decisions based on an on-line algorithm. It periodically monitors the query execution behavior, and finds an optimum ISN time budget and requery threshold configuration to minimize power consumption within the given response time and quality constraints.

6.5.2 TailCut Design Principle

There are three main challenges in designing TailCut: (1) At ISNs, upon receiving a query, how to predict its processing time, (2) how to determine if Re-Query is needed, and (3) how to determine/adjust the ISN time budget dynamically. In this section, we will present a detailed discussion on the design principle of our TailCut scheme.

Query processing time prediction: The score of every document in the postings lists is computed to find the top-K results at each ISN. Thus the query processing time is highly dependent on the length of postings lists. Figure 6.7 shows the relationship between query processing time and the total length of the postings lists. In general, the query processing time increases with the total length of postings lists. To be more precise, at a smaller length, the processing time increases faster with the increase of the length.

As shown in figure 6.7, We find that a piecewise linear regression model can achieve a good approximation of this relationship, especially for queries with a long processing time that we focus on. Note that some existing studies have proposed to include more informa-

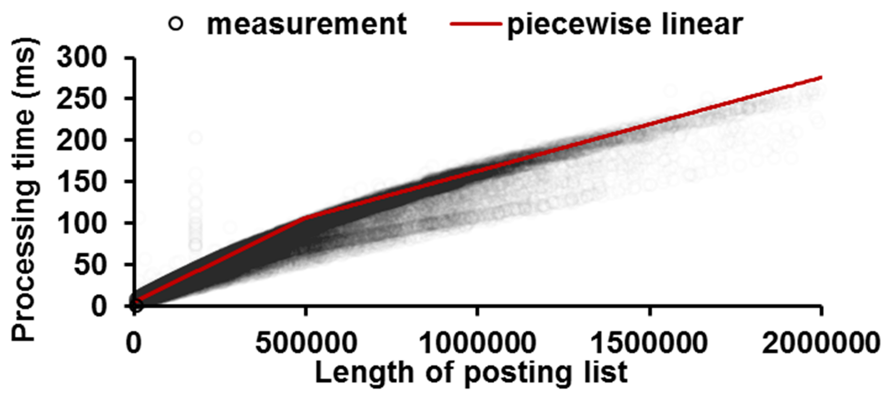


Figure 6.7: Relationship between query processing time and the total length of postings lists.

tion, such as number of query terms and query complexity, to predict the processing time with a better accuracy [65, 73]. Although these prediction algorithms can potentially improve the accuracy, they also introduce additional processing time overhead and complexity. We leave these advanced prediction algorithms as extensions for our future work.

Adjusting ISN time budget: The ISN time budget, T_b, serves as the most important parameter to improve the user query responsiveness and power consumption, yet at the cost of reducing the quality. In general, we want to choose the smallest T_b that can meet the latency and quality constraints and reduce power consumption.

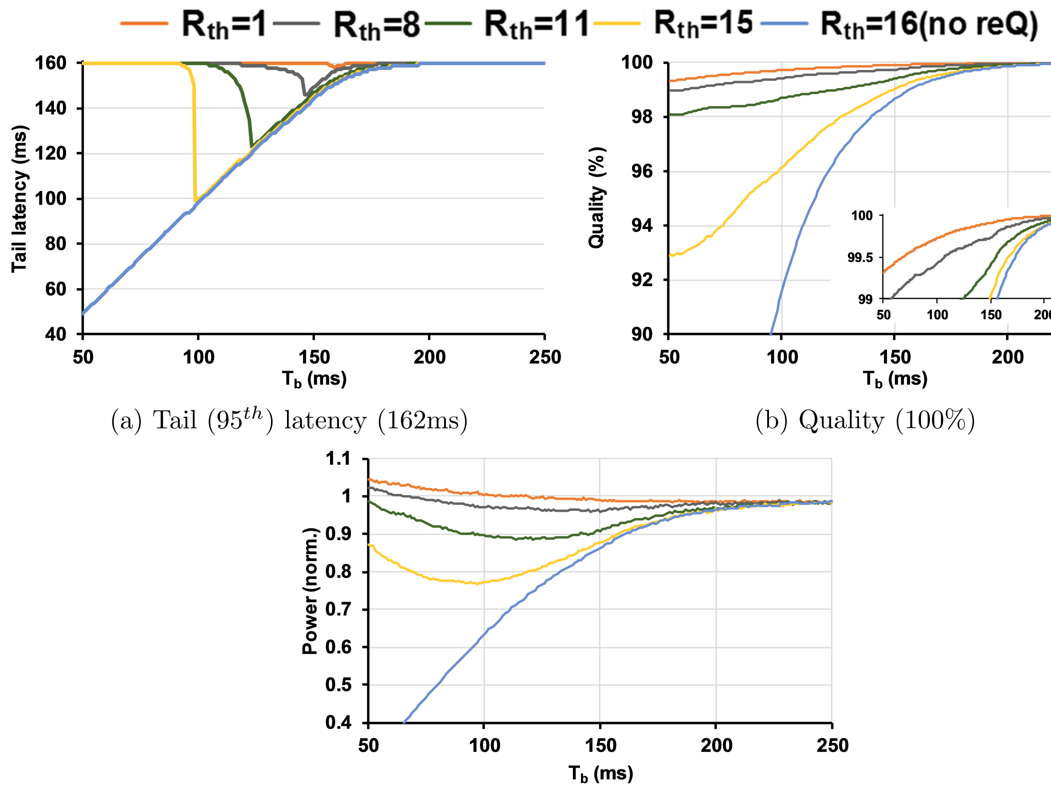
When to send requeries: In the aggregator, the quality inspector will determine whether requeries are needed. It will compare the number of "exceed budget" responses, R_e, to a threshold R_{th}. It sends requeries only when $R_e > R_{th}$, which means that too many ISNs drop their queries due to insufficient time budgets. The value of R_{th} dictates the degree of the trade-off. A higher R_{th} translates to less frequent requeries, which also leads to a lower aggregator response time, ISN power consumption and search result quality. Thus, we need to judiciously choose the requery threshold R_{th} to reduce the energy consumption while meeting the latency and quality constraints.

6.5.3 TailCut Performance Analysis

In this section, we provide detailed performance analysis on our TailCut scheme. The experimental setup is provided later in section 6.7.1, and we only give 5 different values of R_{th} for the simplicity. There are two design parameters determining the performance of the TailCut: ISN time budget (T_b) and requery threshold (R_{th}). Figure 6.8 shows the results of the response quality, response tail latency and average total ISNs power consumption under different T_b and R_{th}. In this figure, $T_b = 250$ms and $R_{th} = 16$ represent the "no ISN time budget" scheme and "no requery (no reQ)" scheme, respectively. Note that the quality and latency performances under no reQ scheme are similar to those under the aggregator timeout scheme.

Reducing tail latency: Clearly, the responsiveness of user queries will be improved by setting a limit on the ISN query processing time. As shown in figure 6.8(a), in the no requery scheme, the tail latency of the user queries is less than T_b. Because all ISN queries predicted to be longer than T_b are dropped, T_b represents the worst case query latency. As a result, the tail latency will be smaller than T_b. On the other hand, with requeries ($R_{th} < 16$), the tail latency will increase when T_b is below a threshold, instead of keeping on decreasing. For example, with $R_{th} = 15$ (requery only when all ISNs predict their queries will spend longer than T_b to process), the tail latency decreases similar to the no-requery scheme as T_b decreases. However, as T_b decreases below a turning point (100ms), the tail latency increases rapidly to 160ms. This is because when T_b is set too low, long queries that are predicted to be dropped will be sent to the ISN again for requery. When the requery frequency is higher than 5%, the tail latency will be determined by the requery latency, which is greater than T_b. Other values of R_{th} have similar impacts, but with different turning points. In summary, in terms of tail latency, there exists an optimum T_b such that the tail latency is minimized, and larger values of R_{th} give a lower tail latency.

Decreased quality: Figure 6.8(b) shows the impact of different design parameters on the search quality. With the no requery scheme, we can see that the quality drops rapidly as T_b decreases due to the increasing number of the dropped ISN queries. On the



(a) Tail (95th) latency (162ms) (b) Quality (100%) (c) Power (1)

Figure 6.8: TailCut performances with different T_b and R_{th}. The baseline system performances are given in parenthesis.

other hand, requeries significantly improve the quality. With requesy sent, a query will have 100% quality. With the monitoring at the aggregator level, requeries can ensure that only a certain portion of ISN queries are dropped. Even with $R_{th} = 15$, which only sends requery when all ISNs predict to drop, the quality improves 65% on average over the no requery scheme. The lower the R_{th}, the higher the quality will be, yet with a marginal improvement.

Saving power: The main advantage of TailCut is the ability to save power. Through query time prediction, the long ISN queries will be dropped, reducing their workloads and power consumption. Figure 6.8(c) shows the average total ISN power consumption with different T_b and R_{th}. The power consumption is normalized to that of the baseline scheme where all ISN queries are processed. Without requeries, all ISN queries longer than T_b are dropped, which translates into a significant power saving. With requeries, fewer ISN queries are dropped, resulting in a increase in power consumption. We see a typical 'U' curve, where the power consumption first drops as T_b decreases and then increases when T_b reach an "optimum" point. As T_b decreases, although more queries are dropped, the possibility of requery also increases. Thus, power saving will increase before the requeries outweigh the dropped ISN queries in terms of energy consumption. Moreover, we can notice that power consumption is even higher than the baseline scheme in some cases, especially with small R_{th}. This is because of the ISN query initialization and prediction overhead. When an ISN receives query requests from aggregator, it does some preprocessing to the request, such as html parsing, handler initialization, etc. An ISN's decision of dropping the query is made after these initialization operations. As a result, the power consumption of initialization is doubled when requeries are sent. With a small R_{th}, this query initialization power leads to the increase in power consumption compared to the baseline scheme.

6.6 Configuration of TailCut

From figure 6.8, we can see that the tail latency, quality and power consumption vary differently with T_b and R_{th}. To have a better understanding of the overall performance of TailCut, we compile the results given in figure 6.8 into a *quality constrained power-latency performance* chart shown in figure 6.9. In this figure, each point represents a pair of power and latency results with a configuration (T_b,R_{th}) that achieves a quality greater than 99%.

The coordinate of a point (x,y) corresponds to the power consumption and tail latency results, respectively. Each line then represents the results with a fixed R_{th} and decreasing T_b, starting from 200ms (right most point). This figure shows that TailCut can achieve 25% latency reduction and 14% power saving with configuration (122,11). Moreover, further power saving can be achieved at the cost of decreased latency reduction.

Figure 6.9 gives some insights on the performance dynamics of TailCut. First, TailCut with requery consistently outperforms the no requery scheme (solid black curve) in terms of both power and latency reduction, which demonstrates the effectiveness of our TailCut design. Second, for a fixed R_{th}, latency and power reduction are "in phase". As T_b decreases, the quality constraint will be violated before the global minimum of latency and power consumption. Combined with the observation that quality is a non-decreasing

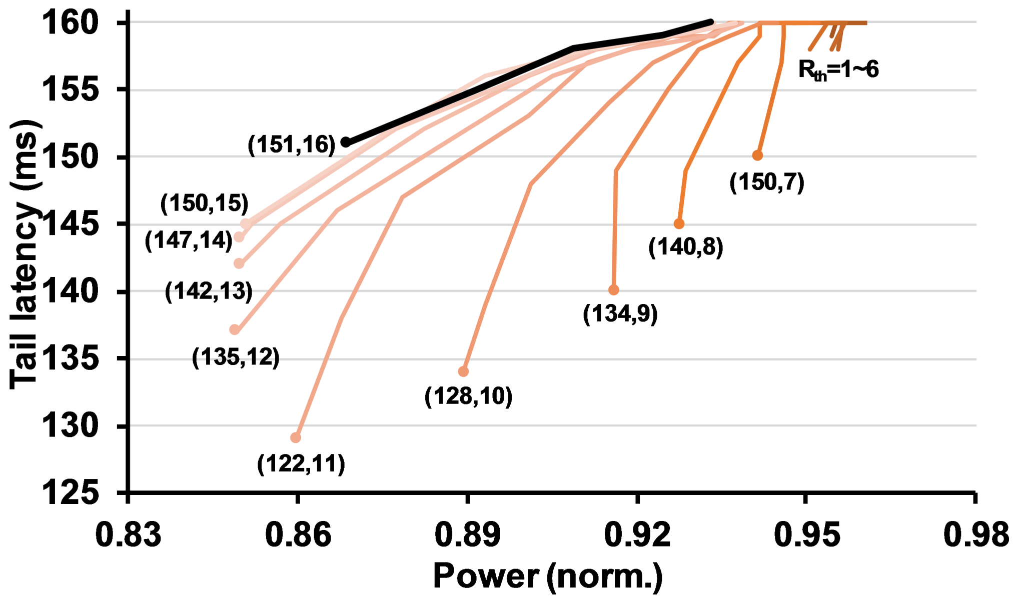


Figure 6.9: Power-latency performance under quality constraint (99%). For each line, T_b is decreasing from 200ms, with $T_b = 200$ms at the right most point of each line

function, the minimum T_b that satisfies the quality constraint will also lead to the minimum latency and power consumption for a given R_{th}. However, the power-latency interaction becomes more complicated with different R_{th}.

As R_{th} increases, the "minimum point" shifts toward the left-bottom corner, meaning that power and tail latency decrease simultaneously. As R_{th} increases beyond 11, the minimum point moves in left-up direction. In this region, there is a trade-off between power and latency. Further increasing R_{th} beyond 14 will increase both latency and power consumption. There is no straight forward relationship between the configuration (T_b,R_{th}) and the performance trade-offs.

While the close-form equation for describing the performance for a given configuration is unattainable, we develop a low-complexity 2-dimensional search algorithm to efficiently find T_b and R_{th} without exhaustively exploring the entire design space. The algorithm can be divided into two iterative steps: (1) find the T_b under a fixed R_{th} (corresponds to finding the end point of a line in figure 6.9), and (2) select the configuration that minimizes power. The details of our algorithm is given in Algorithm 2.

Step 1: Since our goal is to minimize power consumption, by observing figure 6.8(c), we can see that the power consumption for each R_{th} has only one global minimum. By using binary search respect to T_b (inner while loop), we can efficiently find the minimum power consumption for a given R_{th}.

Step 2: As shown in figure 6.9, the minimum point for each R_{th} forms a U curve. Thus, we also use binary search to find the optimum configuration (outer loop).

Acquiring the performance: In our algorithm, we need to evaluate the performance with the different configuration. Without close-form equations, we use a simulator to acquire the performance. The simulator will take the arrival time, processing time and the responses for the ISN queries traces as inputs. These traces are replayed with the configuration to generate the performance metrics.

Algorithm 2 Configuration determination

```
1: Input: query trace, $Q_{th}, L_{th}$ Output: $T_b, R_{th}$
2: $A.left = min(T_b), A.right = max(T_b), A.P_{pre} = \infty$
3: while $A.left < A.right$ do
4:     $A.mid = \lfloor (A.left + A.right)/2 \rfloor$
5:     $B.left = min(R_{th}), B.right = max(R_{th})$
6:     $B.P_{pre} = \infty, B.L_{pre} = L_{th}$
7:     while $B.left < B.right$ do
8:         $B.mid = \lfloor (B.left + B.right)/2 \rfloor$
9:         $(Q_{cur}, L_{cur}, P_{cur})$=simulate($B.mid, A, mid$)
10:        if $P_{cur} < B.P_{pre} \,\&\, Q_{cur} \geq Q_{th} \,\&\, L_{cur} \leq B.L_{pre}$ then
11:            $B.P_{pre} = P_{cur}$, $B.L_{pre} = L_{cur}$
12:            $B.right = B.mid$
13:        else
14:            $B.left = B.mid + 1$
15:        end if
16:    end while
17:    if $A.P_{pre} < B.P_{pre}$ then
18:        $A.P_{pre} = B.P_{pre}$
19:        $A.right = A.mid$
20:    else
21:        $A.left = A.mid + 1$
22:    end if
23: end while
24: $T_b = B.right, R_{th} = A.right$
25: return $(T_b, R_{th})$
```

Runtime adaptation: With ever-changing user queries and the continuously updated ISN indexes, the performance will vary over time. We periodically compute the optimum configuration to adapt to these variations. At the beginning of each epoch, we first start a "logging period". During this period, we do not apply TailCut, and gather all query traces (query arrival and processing time, and ISN responses). These traces are then fed into the simulator when computing the performance under a given configuration. After determining the updated configuration, it is applied until the end of the epoch. Based on our experimental results, we set the length of logging period and epoch to be 3 and 30 minutes, respectively. These values provide a good balance between responsiveness and power saving while query behavior changes.

6.7 Evaluation

6.7.1 Experiment Setup

Application and workload: We use the industrial strength open-source Lucene /Solr search engine similar to the commercial web search engines [37,105,120]. Lucene/Solr provides a convenient setup for the distributed search environment. We build indexes of a 230GB Wikipedia's English XML export [12], which contains all the wikipedia articles/pages. The document indexes are distributed across different index nodes through Lucene/Solr "SPLIT_SHARD" utility. The search queries are generated from the other machine connected through 10G Ethernet. The query terms are randomly generated from luceneutil term base [6].

Simulator: Since the distributed search engine involves multiple physical servers

Table 6.1: Baseline system configuration.

Name	Configuration
number of ISN	16
number of match doc (top k)	20
Tail latency constraint	150ms
Quality constraint	99%
Query per second (QPS)	10

which we cannot access, we evaluate TailCut with an in-house discrete event-based simulator. Our simulator takes the query arrival time, execution time and ISN query result traces as input. We acquire the traces through real machine experiments. For the user query arrival trace, we randomly generate 100K user queries with poisson arrivals. To acquire the query processing time traces, we *sequentially* run each ISN on a real server with the partitioned document index and the same queries arrival trace. Our server has a 12-core 2.7GHz Xeon E5-2697-v2 processors with 256 GB main memory and 1TB HDD. We disable the DVFS to eliminate run-to-run variations. Before running the experiments, we warm up the server long enough to ensure all indexed data are cached in the main memory. During each run, the length of postings lists, query processing time and the top-K matched documents for each query are logged. Note that the query processing time is measured from the beginning and the end of each query processing, excluding the queuing delay. The gathered traces are then fed into the simulator with TailCut implementation, as shown in figure 6.6, with one aggregator and multiple ISNs. Each ISN are modeled to have 12 CPU cores following our server configuration. In our simulation, a CPU core will spend full power (3.4W) during its query processing, and no power during idle. We then replay the traces with different distributed search system configuration to obtain the latency, quality and power consumption performances.

6.7.2 Results

In this section, we present sensitivity studies on different system configuration to show the effectiveness of our proposed TailCut design. The baseline system configuration is given in table 6.1. The default distributed search system (without any time budget at the aggregator and ISNs) with the baseline configuration will have 162ms tail (95%ile) latency, 100% result quality and no power saving. We vary one configuration at a time to show the impact on the distributed search systems.

Processing time prediction model: The keystone of TailCut is the ability to predict the query processing time at each ISN. In figure 6.7, we show the relationship between the total length of postings lists and the query processing time, and use piecewise linear regression (PWL) to build our prediction model. To understand how the prediction model affects TailCut, we conduct experiments with different prediction models, and give the results in figure 6.10. We compare the results of using (1) linear regression, (2) piecewise linear regression with different number of pieces, and (3) oracle scheme where the query processing time is known beforehand. We can see a significant performance improvement of using PWL model over simple linear model. Obviously, the non-linear relation between postings lists length and query processing time can be better capture by PWL. Also, we can see that increasing the degree of PWL gives a marginal performance improvement. From degree 2 to 8, the power saving only improves by 1% but significantly increases the complexity in piecewise linear regression modeling. Comparing with the oracle scheme, simple PWL model can achieve power saving within a 2% range without complicated prediction algorithms.

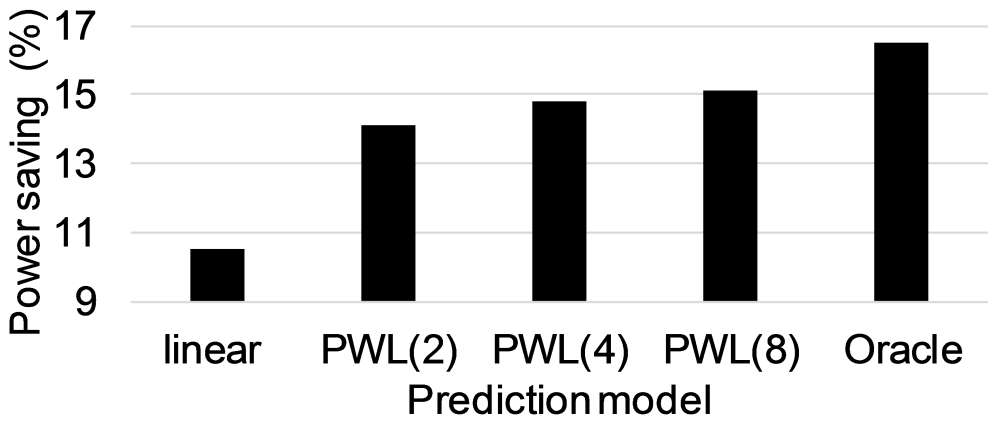


Figure 6.10: TailCut performance under different query processing time prediction model.

Quality and tail latency constraints: The distributed search systems are widely applied to different services, such as web search, product recommendation, advertising, etc., and its performance requirements vary among different service providers. For example, the product recommendation service might require higher result quality for better revenue, but can tolerate longer response time. A web search system might need to balance the answer quality and response time, since both will affect the user experiences hence the revenue. To evaluate how TailCut reacts to different requirements, we perform sensitivity studies on both quality and tail latency constraints and results are shown in figure 6.11.

First, we can see that TailCut is more sensitive to the latency constraint than to the quality constraint. With a fix latency constraint, the achievable power saving decreases almost linearly as quality constraint decreases. On the other hand, as latency constraint decreases with the same quality constraint, the power saving decreases exponentially. This is expected since small T_b and R_{th} will affect more in latency than quality as shown figure 6.8. With 70ms latency constraint, TailCut is only effective when the quality constraint is lower than 95%. On the other hand, TailCut can save power under 150ms latency constraint and

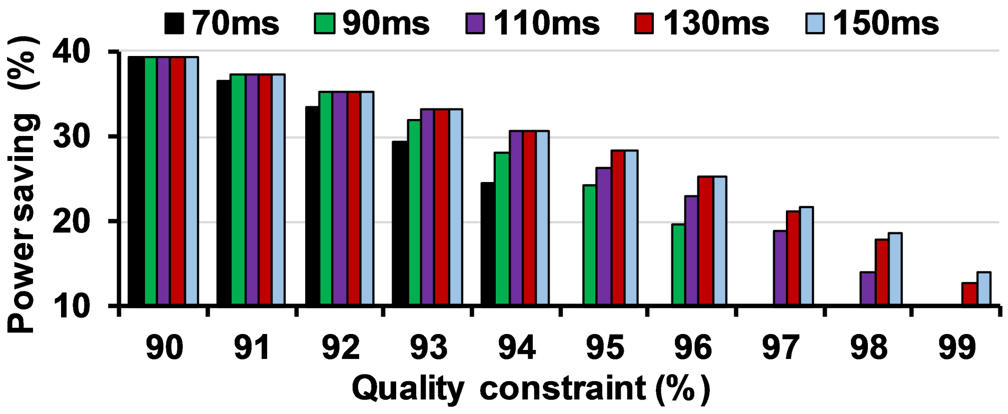


Figure 6.11: TailCut power saving under different quality and latency constraints.

99% quality constraint. In order to achieve high quality results, more ISN queries should be processed. In contrast, reducing latency requires more ISN query dropping. This trade-off between latency and quality determines the power saving performance of TailCut.

System load: While query processing time is an important factor, query queuing behavior also plays an important role in user query responsiveness, especially in high system loads. Figure 6.12 shows the TailCut power saving performance under different system loads.

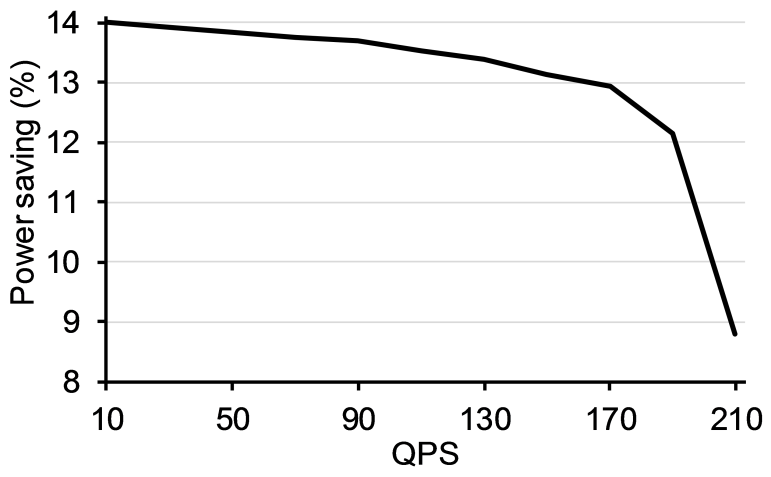


Figure 6.12: TailCut power saving under different load (QPS).

We vary the query-per-second (QPS) from 10 to 210. The power saving slightly decreases as the load increases, from 14% to 13% as QPS increases from 10 to 170. However, the power saving quickly drops after further increasing QPS, only 8.8% at QPS=210. Because of the queuing delay, the configuration update algorithm will choose a *lower* T_b, which aggressively drops ISN queries for satisfying the latency constraint. However, to meet the quality constraint, requries will be sent frequently, resulting in a low power saving. Note that although not shown in this figure, TailCut can also improve the throughput of the system. In the default system (without any time budget), the average utilization of the system is about 80%. With TailCut, the utilization is reduced to 72% because of the ISN queries dropping.

Different number of matched documents: Different matched documents will also affect the performance of TailCut. As shown in figure 6.13, the power saving decreases as the number of requested documents (top-K) increases. This is due to the change of response quality of ISNs. As K increases, it is less likely that an ISN will have no or little

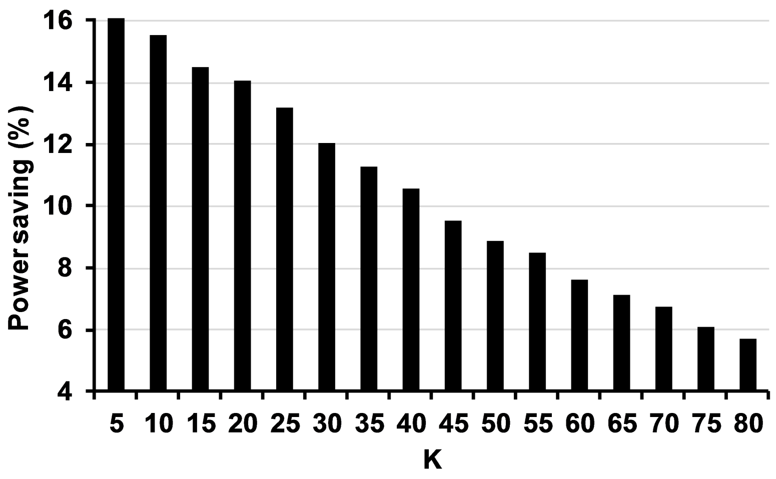


Figure 6.13: TailCut power saving under different matched documents (Top-K).

contribution to the final response, hence dropping a ISN query will have greater impact on the quality. As a result, fewer ISN queries can be dropped with little quality degradation.

Scalability: To see the scalability of our TailCut design, we conduct experiments with different numbers of ISNs and show the results in figure 6.14. In these experiments, for the ISN number larger than 16, we use synthetic ISN query processing time generated based on the correlation shown in figure 6.3. We can see that power saving achieved by TailCut "increases" as the number of ISN increases. With a larger ISN count, it is more likely that an ISN will have no contribution. From the law of total probability, more percentage of ISN queries generated from a user query can be dropped under the same quality constraint, hence higher power saving is achieved. Although increasing, this power saving will asymptotically approach a upper bound (~26.5%) with further increasing number of ISN, and this value will be affected by the selection of quality and latency constraints.

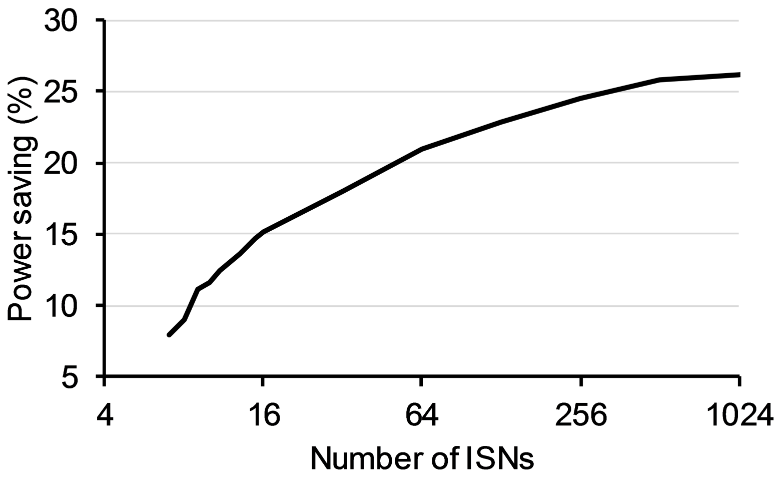


Figure 6.14: TailCut power saving under different number of ISNs.

Chapter 7

Conclusions

The explosive growth of network and data center applications require orders-of-magnitude increase in network bandwidth and throughput. The delay-sensitive interactive services, such as web search, online recommendations, gaming, and multimedia streaming applications, have become an integral part of our lives and constitute an increasingly high portion of network and data center workloads. To attract users and generate revenue, these services require high-quality and timely responses, placing a high priority on minimizing latency. As the service providers aggressively scale the infrastructure capability by employing high-performance computers, high power consumption poses a significant challenge to scalable system design. Higher power consumption not only increases the cost of operation, it also increases core temperature, which exponentially increases the cost of cooling and packaging, as well incurs indirect and life-cycle costs due to reduced system performance, circuit reliability and chip lifetime. As a result, thermal awareness is required in the system design. In this thesis, we focused on developing power and thermal management techniques for network and data center applications while considering various performance constraints. Our goal is to reduce the power/energy consumption while satisfying not only throughput, but also the thermal, latency, and quality constraints.

In chapter 3, we have presented a Vacation scheme, a thermal management technique for network packet processing on multicore architecture. By introducing forced working and vacation periods, we minimize the peak core temperature and power consumption with the use of the C-states provided by OS without serious latency penalty. To understand and predict the performance under Vacation scheme, we proposed novel performance models for the temperature rise, power consumption and average latency with equivalent RC thermal modeling and vacation queuing theory. On top of our per-core vacation scheme, we proposed a heterogeneous load distribution, which brings higher achievable load and power saving while maintaining peak core temperature under the given thermal constraint. We have demonstrated the effectiveness of our approach with the real multicore server running CRC, a packet processing application, and traffic trace. The experimental results show that our vacation scheme can sustain 10% more load on an average without thermal constraint violation compared with the state-of-the-practice thermal management schemes. Also, under the thermal constraint, our vacation scheme can achieve up to 25% power saving.

Then, in chapter 4, we have presented a power management technique for multicore, called smart sleep scheme, for network and datacenter applications. First, we identify the causes for major power wastage during the idle period. We minimize the power consumption by reducing idle period power consumption through per-core sleep, which effectively reshapes the processing pattern and utilizes the existing C-states for power saving.

Considering the tail latency constraint for the data center applications, we derived a novel performance model for the probability distribution function of the latency under our smart sleep scheme, and validated with experimental results. Furthermore, we consider core sizing along with the per-core sleep, which brings higher power saving while satisfying traffic load demands and QoS requirement. We demonstrated the effectiveness of our approach with the real multicore server running three network packet processing applications and a latency sensitive data center application with both synthetic and real-world traffic traces. The experimental results show that our proposed technique can achieve up to 44% power saving and 60% energy proportionality increase compared with the state-of-the-practice power management technique under real world traffic traces.

Next, we explored the implications of short-term traffic variation existing in the data center. In chapter 5, we first presented DynSleep, a fine-grain power management technique that reduces core power consumption by intelligently determining the CPU sleep period at per-request basis and utilizing CPU deep sleep states. DynSleep uses a novel performance model to account for the uncertainties, such as random request arrivals and service time, in the latency-critical applications. We implemented DynSleep with Memcached, and demonstrated the effectiveness of our approach with the real multicore server. The experimental results show that our DynSleep can achieve up to 65% power saving. Then, we performed a detailed analysis on the state-of-the-art techniques for both sleep and DVFS based power management scheme and found out that the existing dynamic power management techniques are largely ineffective because of the high state transition overhead, short request service time and strict tail latency constraint. Our analysis also shows that sleep or DVFS alone can not fully exploit the latency slack for power saving. Based on the observations, we designed μDPM, an all encompassing power management framework that coordinates service delay, per-core sleep states, voltage frequency scaling and load distribution. μDPM extends our dynamic sleep scheme by taking into account the request processing speed. Also, we developed a novel request redirection algorithm, which aims to minimize both sleep and DVFS state transitions, which are the major source of energy inefficiency while applying sleep and DVFS power management. The simulation results show that μDPM reduces processor energy consumption by up to 32%, and consistently outperforms state-of-the-art techniques by 2x while satisfying strict SLA constraint. Contrary to existing assumptions that sleep states cannot be used in latency-critical workloads, μDPM demonstrates significant energy savings even with microsecond service time and latency constraint.

Lastly, we look into the system wide datacenter performances for online data-intensive (OLDI)applications. The service provider often sets strict constraint on the tail latency and response quality for these applications. In chapter 6, we explored the complex trade-off among latency, quality and energy consumption in the distributed search system, and gained some insights through detailed analysis of the workload characteristics. We proposed TailCut, an aggregation-ISN cooperative policy, to reduce power consumption subject to tail latency and response quality constraints. In TailCut, the ISNs judiciously discards query executions based on our predictor, and the aggregator issues the requery to ISNs if the quality constraint is not met. With the detailed performance analysis of TailCut, we developed an online algorithm to quickly configure TailCut — time budget and requery threshold. We conducted experiments using execution traces logged from deploying Lucene/Solr on a real server with the full Wikipedia dataset. Our experimental results show that TailCut can achieve up to 39% power saving, while satisfying the tail latency and quality constraints.

Some potential future research directions are listed below. First, this thesis focused on the power and thermal management of the server for network and data center applications without touching other parts of the data center. Specifically, network power saving should also be considered. It will be challenging to incorporate various techniques considering the performance interaction between network and server. Second, it would be interesting combine the techniques presented in chapter 5 and 6, as dropping query processing should provide more latency slack for the server and create more opportunities for DVFS and CPU sleep to save power. Third, this thesis only focused on single application, query processing in the data center might consist of different stages. Multiple applications, such as data analysis, data caching and search, work together to serve a user request. Each application has different workload characteristic and behavior, and might have dependency with other applications. It would be interesting to explore the interaction among different applications with the different SLA constraints.

Bibliography

[1] Apach solr search engine. http://lucene.apache.org/solr/.

[2] Caida equinix. http://www.caida.org/.

[3] The idle governor in linux kernel. http://www.kernel.org.

[4] Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm.

[5] Intel idle driver for linux. http://lxr.free-electrons.com/source/drivers/idle/intel_idle.c.

[6] Luceneutil. http://github.com/mikemccand/luceneutil.

[7] Memcached. http://memcached.org/.

[8] pytimechart. http://packages.python.org/pytimechart/.

[9] Receive side scaling. https://docs.microsoft.com/zh-tw/windows-hardware/drivers/network/ndis-receive-side-scaling2.

[10] Specjbb. http://www.spec.org/jbb2013/.

[11] Thermal throttling in linux kernel. http://www.kernel.org.

[12] Wikipedia database. https://en.wikipedia.org/wiki/Wikipedia:Database_download.

[13] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Paramvir Bahl, and Rajesh Gupta. Somniloquy: Augmenting network interfaces to reduce pc energy usage. In *NSDI*, volume 9, pages 365–380, 2009.

[14] M. Alian, A. H. M. O. Abulila, L. Jindal, D. Kim, and N. S. Kim. Ncap: Network-driven, packet context-aware power management for client-server architecture. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[15] Vlasia Anagnostopoulou, Susmit Biswas, Heba Saadeldeen, Alan Savage, Ricardo Bianchini, Tao Yang, Diana Franklin, and Frederic T. Chong. Barely alive memory servers: Keeping data active in a low-power state. *J. Emerg. Technol. Comput. Syst.*, 8(4):31:1–31:20, November 2012.

[16] Murali Annavaram. A case for guarded power gating for multi-core processors. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 291–300. IEEE, 2011.

[17] Raid Ayoub, Krishnam Indukuri, and Tajana Simunic Rosing. Temperature aware dynamic workload scheduling in multisocket cpu servers. *IEEE transactions on Computer-aided design of integrated circuits and systems*, 30(9):1359–1372, 2011.

[18] Claudine Santos Badue, R Baeza-Yates, B Ribeiro-Neto, Artur Ziviani, and Nivio Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Information processing & management*, 2007.

[19] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: processor-level preventive thermal management via idle cycle injection. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 89–94. IEEE, 2011.

[20] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[21] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[22] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 2007.

[23] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[24] Alexander Branover, Denis Foley, and Maurice Steinman. Amd fusion apu: Llano. *Ieee Micro*, 32(2):28–37, 2012.

[25] Sergey Brin and Lawrence Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 2012.

[26] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. *CIKM*, 2003.

[27] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182. IEEE, 2001.

[28] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. Fivrfully integrated voltage regulators on 4th generation intel® core socs. In *Applied Power Electronics Conference and Exposition (APEC), 2014 Twenty-Ninth Annual IEEE*, pages 432–439. IEEE, 2014.

[29] M Cho, N Sathe, M Gupta, S Kumar, S Yalamanchilli, and S Mukhopadhyay. Proactive power migration to reduce maximum value and spatiotemporal non-uniformity of on-chip temperature distribution in homogeneous many-core processors. In *Semiconductor Thermal Measurement and Management Symposium, 2010. SEMI-THERM 2010. 26th Annual IEEE*, pages 180–186. IEEE, 2010.

[30] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 international symposium on Low power electronics and design*, pages 213–218. ACM, 2007.

[31] Chih-Hsun Chou and Laxmi N Bhuyan. Thermal-aware vacation and rate adaptation for network packet processing. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 185–196. ACM, 2014.

[32] Chih Hsun Chou and Laxmi N Bhuyan. A multicore vacation scheme for thermal-aware packet processing. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 565–572. IEEE, 2015.

[33] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. Dynsleep: Fine-grained power management for a latency-critical data center application. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 212–217, New York, NY, USA, 2016. ACM.

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

[35] Matthew Curtis-Maury, James Dzierwa, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 157–166. ACM, 2006.

[36] Jeffrey Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM.

[37] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 2013.

[38] Qian Diao and Justin Song. Prediction of cpu idle-busy activity pattern. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 27–36. IEEE, 2008.

[39] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. A candidate filtering mechanism for fast top-k query processing on modern cpus. *SIGIR*, 2013.

[40] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. *SIGIR*, 2011.

[41] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[42] Zhihui Du, Hongyang Sun, Yuxiong He, Yu He, David A Bader, and Huazhe Zhang. Energy-efficient scheduling for best-effort interactive services to achieve high response quality. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 637–648. IEEE, 2013.

[43] Nosayba El-Sayed, Ioan A Stefanovici, George Amvrosiadis, Andy A Hwang, and Bianca Schroeder. Temperature management in data centers: Why some (might) like it hot. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):163–174, 2012.

[44] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[45] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 175–184, New York, NY, USA, 2006. ACM.

[46] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[47] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[48] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael A Kozuch. Optimality analysis of energy-performance trade-off for server farm management. *Performance Evaluation*, 67(11):1155–1171, 2010.

[49] Mohamed Gomaa, Michael D Powell, and TN Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In *ACM SIGARCH Computer Architecture News*, volume 32, pages 260–270. ACM, 2004.

[50] Flavius Gruian and Krzysztof Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 449–455. ACM, 2001.

[51] Danhua Guo, Laxmi Narayan Bhuyan, and Bin Liu. An efficient parallelized l7-filter design for multicore servers. *IEEE/ACM Transactions on Networking*, 20(5):1426–1439, 2012.

[52] Vinay Hanumaiah, Ravishankar Rao, Sarma Vrudhula, and Karam S Chatha. Throughput optimal task allocation under thermal constraints for multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference*, pages 776–781. ACM, 2009.

[53] Vinay Hanumaiah and Sarma Vrudhula. Temperature-aware dvfs for hard real-time applications on multicore processors. *IEEE Transactions on Computers*, 61(10):1484–1494, 2012.

[54] Md E Haque, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *ACM SIGPLAN Notices*, 50(4):161–175, 2015.

[55] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. Zeta: Scheduling interactive services with partial execution. *SOCC*, 2012.

[56] Yuxiong He, Sameh Elnikety, and Hongyang Sun. Tians scheduling: Using partial processing in best-effort applications. *ICDCS*, 2011.

[57] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA-21, Washington, DC, USA, 2015. IEEE Computer Society.

[58] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. pages 32–37, 2004.

[59] Muhammad Faisal Iqbal and Lizy K John. Efficient traffic aware power management for multicore communications processors. In *Architectures for Networking and Communications Systems (ANCS), 2012 ACM/IEEE Symposium on*, pages 123–133. IEEE, 2012.

[60] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture*, pages 347–358. IEEE Computer Society, 2006.

[61] 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*, pages 359–370. IEEE Computer Society, 2006.

[62] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. *ACM SIGARCH Computer Architecture News*, 2010.

[63] Keon Jang, Sangjin Han, Seungyeop Han, Sue B Moon, and KyoungSoo Park. Sslshader: Cheap ssl acceleration with commodity processors. In *NSDI*, 2011.

[64] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 155–168. ACM, 2013.

[65] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. *SIGIR*, 2014.

[66] Anil Kanduri, Amir M Rahmani, Pasi Liljeberg, Ahmed Hemani, Axel Jantsch, and Hannu Tenhunen. A perspective on dark silicon. In *The Dark Side of Silicon*, pages 3–20. Springer, 2017.

[67] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 158–169. IEEE, 2015.

[68] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 31–40. IEEE, 2014.

[69] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 598–610. ACM, 2015.

[70] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM.

[71] Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety. Measuring and managing answer quality for online data-intensive services. *ICAC*, 2015.

[72] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *IEEE computer*, 36(12):68–75, 2003.

[73] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. *WSDM*, 2015.

[74] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134. IEEE, 2008.

[75] Joonho Kong, Sung Woo Chung, and Kevin Skadron. Recent thermal management techniques for microprocessors. *ACM Computing Surveys (CSUR)*, 44(3):13, 2012.

[76] Robin Kravets and Parameshwaran Krishnan. Application-driven power management for mobile communication. *Wireless Networks*, 6(4):263–277, 2000.

[77] J Kuang and L Bhuyan. Thermal-aware scheduling of network applications on multi-core architecture. *Proceedings of the 14th ACM/IEEE ANCS*, pages 1–14, 2009.

[78] Jilong Kuang and Laxmi Bhuyan. Optimizing throughput and latency under given power budget for network packet processing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[79] Jilong Kuang and Laxmi Bhuyan. Predictive model-based thermal management for network applications. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 57–68. IEEE, 2011.

[80] Jilong Kuang, Laxmi Bhuyan, and Raymond Klefstad. Traffic-aware power optimization for network applications on multicore servers. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1006–1011. ACM, 2012.

[81] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[82] Jungseob Lee and Nam Sung Kim. Optimizing throughput of power-and thermal-constrained multicore processors using dvfs and per-core power-gating. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 47–50. IEEE, 2009.

[83] Kin K Leung and Martin Eisenberg. A single-server queue with vacations and non-gated time-limited service. *Performance Evaluation*, 12(2):115–125, 1991.

[84] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[85] Yanpei Liu, Stark C. Draper, and Nam Sung Kim. Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers. pages 313–324, 2014.

[86] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 301–312, Piscataway, NJ, USA, 2014. IEEE Press.

[87] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015.

[88] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. Dynamic power gating with quality guarantees. pages 377–382, 2009.

[89] Yadi Ma, Suman Banerjee, Shan Lu, and Cristian Estan. Leveraging parallelism for multi-dimensional packetclassification on software routers. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 227–238. ACM, 2010.

[90] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[91] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 205–216, New York, NY, USA, 2009. ACM.

[92] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[93] David Meisner and Thomas F. Wenisch. Dreamweaver: Architectural support for deep sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 313–324, New York, NY, USA, 2012. ACM.

[94] David Meisner, Junjie Wu, and Thomas F Wenisch. Bighouse: A simulation infrastructure for data center systems. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 35–45. IEEE, 2012.

[95] Gokhan Memik, William H Mangione-Smith, and Wendong Hu. Netbench: A benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42. IEEE Press, 2001.

[96] Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. A study of thread migration in temperature-constrained multicores. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(2):9, 2007.

[97] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[98] Alexander W Min, Ren Wang, James Tsai, Mesut A Ergin, and Tsung-Yuan Charlie Tai. Improving energy efficiency for mobile platforms by exploiting low-power sleep states. In *Proceedings of the 9th conference on Computing Frontiers*, pages 133–142. ACM, 2012.

[99] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9), 2014.

[100] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI*, volume 8, pages 323–336, 2008.

[101] Terry Nelms and Mustaque Ahamad. Packet scheduling for deep packet inspection on multi-core architectures. In *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*, pages 1–11. IEEE, 2010.

[102] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[103] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do nothing, efficiently. In *Proceedings of the Linux Symposium*, volume 2, pages 119–125. Citeseer, 2007.

[104] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 1996.

[105] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S McKinley. Exploiting processor heterogeneity in interactive services. *ICAC*, 2013.

[106] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Ieee micro*, 32(2):20–27, 2012.

[107] Marcus T Schmitz, Bashir M Al-Hashimi, and Petru Eles. Iterative schedule optimization for voltage scalable distributed embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):182–217, 2004.

[108] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *velocity web performance and operations conference*, 2009.

[109] Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Giovanni F Del Nero, Donatella Sciuto, and Marco D Santambrogio. Thermos: System support for dynamic thermal management of chip multi-processors. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 41–50. IEEE, 2013.

[110] Kevin Skadron, Tarek Abdelzaher, and Mircea R Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 17–28. IEEE, 2002.

[111] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. The case for lifetime reliability-aware microprocessors. In *ACM SIGARCH Computer Architecture News*, volume 32, page 276. IEEE Computer Society, 2004.

[112] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *ACM Sigplan Notices*, volume 45, pages 269–280. ACM, 2010.

[113] Naishuo Tian and Zhe George Zhang. *Vacation queueing models: Theory and Applications*, volume 93. Springer Science & Business Media, 2006.

[114] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 585–597, New York, NY, USA, 2015. ACM.

[115] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *ACM SIGARCH computer architecture news*, volume 37, pages 314–324. ACM, 2009.

[116] Zhe Wang and Sanjay Ranka. A simple thermal model for multi-core processors and its application to slack allocation. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[117] Daniel Wong and Murali Annavaram. Knightshift: Scaling the energy proportionality wall through server-level heterogeneity. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 119–130. IEEE, 2012.

[118] Daniel Wong and Murali Annavaram. Implications of high energy proportional servers on cluster-wide energy proportionality. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 142–153. IEEE, 2014.

[119] Inchoon Yeo, Chih Chun Liu, and Eun Jung Kim. Predictive dynamic thermal management for multicore systems. In *Proceedings of the 45th annual Design Automation Conference*, pages 734–739. ACM, 2008.

[120] Jeong-Min Yun, Yuxiong He, Sameh Elnikety, and Shaolei Ren. Optimal aggregation policy for reducing tail latency of web search. *SIGIR*, 2015.

[121] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda. Carb: A c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters*, PP(99):1–1, 2016.

[122] Xin Zhan, Reza Azimi, Svilen Kanev, David Brooks, and Sherief Reda. Carb: A c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters*, 2016.

[123] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 456–468. IEEE, 2016.

[124] Wenli Zheng and Xiaorui Wang. Data center sprinting: Enabling computational sprinting at the data center level. *ICDCS*, 2015.