UNIVERSITY OF CALIFORNIA
RIVERSIDE

Energy Saving in Data Centers through Traffic and Application Scheduling

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

Doctor of Philosophy

in

Computer Science

by

Liang Zhou

September 2020

Dissertation Committee:

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

The Dissertation of Liang Zhou is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Finishing this dissertation ends one major chapter of my life and triggers the beginning of another. 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.

First, I would like to express my deepest gratitude to my advisor, Dr. Laxmi N. Bhuyan, for his excellent guidance and providing me with a pleasant atmosphere for doing research over the past five years. I am also very grateful to Dr. K. K. Ramakrishnan for his previous contribution and endless help in my research. I also want to thank Dr. Shaolei Ren and Dr. Daniel Wong for serving on my dissertation committee. Their constructive suggestions helped improve the quality of this dissertation.

In addition to the support from academia, I also earned many helps from my intern mentors at Facebook: Guangdeng Liao and Jeevan Shankar. They are patient and are willing to share what they know with me. My internship not only broadened my horizon, but also laid a good foundation for my career.

I would also like to thank former and current members from my lab, Mehmet Esat Belviranli, Chih-Hsun Chou, Devashree Tripathy, Hadi Zamani, Sourav Panda and Quan Fan, for their help during my stay at UCR and valuable discussions on my research.

Finally, I would like to thank my parents and the entire family. They were always supporting me and encouraging me with their best wishes. Without their support, I could not have reached this point.

This dissertation includes content published in the following proceedings:

1. Liang Zhou, Chih-Hsun Chou, Laxmi N. Bhuyan, K. K. Ramakrishnan, Daniel Wong,"Joint Server and Network Energy Saving in Data Centers for Latency-Sensitive Applications", in Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018.

2. Liang Zhou, Laxmi N. Bhuyan, K. K. Ramakrishnan,"DREAM: DistRibuted Energy-Aware traffic Management for Data Center Networks", in Proceedings of the Tenth ACM International Conference on Future Energy Systems (e-Energy), 2019.

3. Liang Zhou, Laxmi N. Bhuyan, K. K. Ramakrishnan,"Goldilocks: Adaptive Resource Provisioning in Containerized Data Centers", in Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS), 2019.

4. Liang Zhou, Laxmi N. Bhuyan, K. K. Ramakrishnan,"Swan: A Two-Step Power Management for Distributed Search Engines", in Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2020.

5. Liang Zhou, Laxmi N. Bhuyan, K. K. Ramakrishnan,"Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines", in Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.

To my parents and the entire family.

ABSTRACT OF THE DISSERTATION

Energy Saving in Data Centers through Traffic and Application Scheduling

by

Liang Zhou

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2020
Dr. Laxmi N. Bhuyan, Chairperson

Energy saving in data centers is challenging due to workload fluctuation and strict application latency constraints. In this dissertation, we aim to develop a suite of practical techniques to improve the energy efficiency of data centers, ensuring that they are effective, scalable and responsive. We have studied this research topic from two perspectives: server and network load consolidation *and* scheduling of latency-sensitive applications. All our techniques have been experimentally shown to produce more energy savings compared to state-of-the-art techniques.

It is known that servers consume maximum energy in a data center. We propose Goldilocks, a resource provisioning system for optimizing both power and task completion time by allocating tasks to servers in groups. Tasks hosted in containers are grouped together by running a graph partitioning algorithm that considers the frequency of communication between the containers. The groups are allocated to a subset of the servers near to each other and all idle servers are turned off to save power. As a next step, we make the data center network energy proportional by developing a distributed traffic consolidation

scheme named DREAM. DREAM splits a TCP flow across multiple paths by adapting the path selection probability for sending a flowlet. The unused switches and links are disabled to save power.

Several techniques have been developed in the literature to save energy on servers through Dynamic Voltage and Frequency Scaling (DVFS) and sleep-based schemes. We have developed EPRONS to minimize the overall data center's power consumption by trading-off network slack in favor of providing additional slack by using DVFS. We design a two-step DVFS scheme to save more power for latency-sensitive applications. Since accurate estimation of the latency is difficult, we extend the two-step DVFS with a neural network based predictor to judiciously boost the CPU frequency at the right time to catch-up to the deadline. Finally, in addition to optimizations at the Index Serving Node (ISN), we design a coordinated time budget assignment framework between the aggregator and ISNs in a search engine to improve search quality and latency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large data centers hosting tens of thousands of servers and networking devices have increasingly become society's core information infrastructure supporting a myriad of latency-critical services, including the most widely used web search [98, 43, 32]. Typically, data centers are over-provisioned so as to satisfy application's Service Level Agreements (SLAs) at peak loads. Servers in data centers usually operate at 20-30% utilization [74, 60, 69] and the network link utilizations are around 10% [93, 49]. Running servers and the Data Center Network (DCN) at such low utilizations wastes power [49]. In 2014, data centers in the U.S. consumed 70 billion kWh of energy, representing 1.8% of the U.S. electricity consumption [106]. Data centers can be made energy proportional through two major approaches: load (server and network) consolidation and application scheduling. Although these two research directions have received a great deal of attentions in the recent past, there still remain a number of major limitations in state-of-the-art energy management approaches, with respect to effectiveness, scalability and responsiveness [131, 129].

In this dissertation, we address the shortcomings of state-of-the-art schemes and propose a suite of practical power management techniques for data centers. Our goal is to make the data center energy efficient without introducing additional performance overheads. In this research context, we first develop a graph-based task consolidation on servers and distributed multi-path traffic consolidation in the DCN that outperform existing load consolidation frameworks for energy saving and request latency reduction [129, 131]. We study this problem in the context of tasks running in containers, such as Docker or Linux Containers. Second, this dissertation demonstrates that a holistic system perspective is needed to take advantage of the latency slack offered by a high performance network to utilize a Dynamic Voltage and Frequency Scaling (DVFS) scheme on servers. In addition to trading-off the DCN energy saving and network slack, we show that it is critical to have a two-step DVFS enhanced with neural network predictors for meeting the challenge of inaccurate service time predictions. A coordinated time budget assignment framework is also developed at the aggregator of distributed search engines to determine each request's deadline at ISNs, as well as the ISN cutoff. Specifically, this dissertation makes the following contributions.

Contribution 1: Graph-based Container Placement on Servers. Servers account for 70%-80% of a data center's total energy consumption [49, 116]. Consolidating server resources to save energy in data centers has proved to be challenging due to the variability in the workload [96] and having to meet strict SLAs [108]. Current server consolidation schemes such as Borg [111] and RC-Informed [27] pack *stand-alone* tasks in containers or Virtual Machines (VMs) into a few highly utilized servers without violating

the resource constraints. They aim to improve either the power consumption [111, 27, 112] or task completion time [8], but are unable to achieve improvements in both dimensions. What is more important, a VM in data centers potentially communicates with a very large number of other VMs. For example, in a representative trace from Microsoft for search, the average number of distinct connections per VM is 45 [6]. The rich interactions between VMs makes it difficult for the above frameworks to use locality-aware placement policies [34, 127, 76]. A holistic approach wisely packing *group of tasks* is needed to strike a balance between the power consumption and application's performance.

In this dissertation, we present a graph-based approach, Goldilocks, to elegantly solve the complex resource 'right sizing' problem in data centers to optimize both power and task completion time. Goldilocks uses containers (as an example) to host tasks instead of VMs as they are more light-weight and flexible [124, 30]. Unlike placing stand-alone containers in prior works [102, 127, 111, 112, 8, 27], Goldilocks partitions groups of containers before assigning the container groups to the servers. By running the recursive graph bipartitioning algorithm [77] with the min-cut objective function, containers with high communication frequency are grouped together. We also leverage new findings on power consumption of modern-day servers to ensure that their utilizations are in a range where they are power-proportional. Both testbed implementation measurements and large-scale trace-driven simulations prove that Goldilocks outperforms all the previous works on data center power saving. Goldilocks saves power by 11.7%-26.2% depending on the workload, whereas the best of the implemented alternatives, Borg, saves 8.9%-22.8%. The energy per request for the Twitter content caching workload in Goldilocks is only 33% of RC-Informed. Fi-

nally, the best alternative in terms of task completion time, E-PVM [8], has 1.17-3.29 times higher task completion times than Goldilocks across different workloads.

Contribution 2: Distributed Traffic Consolidation in the DCN. With energy efficient servers [60, 68, 108], the fraction of energy consumed by the networking components can reach 50% in data centers [2], especially when the server utilizations are low, at 15%. Hence, it is desirable to design practical techniques for the energy-proportional DCNs. Traffic consolidation [49, 113, 109] proactively shifts all the traffic to a minimal subset of network devices and completely turns off unused links and switches. State-of-the-start traffic consolidation approaches, such as ElasticTree [49] and CARPO [113], perform a centralized optimization periodically to decide the set of active switches and links based on current traffic demands. Typically, they formulate the energy saving in DCNs as a linear programming model, which usually has high computational complexity. For example, it may take hours for the centralized linear programming model to find the minimal subset of active switches and links in large DCNs [109, 49]. The high computational complexity of centralized traffic consolidation results in energy inefficiency or link congestion, especially when the traffic bursts occur. For improved responsiveness and thereby higher energy savings, traffic consolidation in the DCN should be conducted in a distributed way.

In this research, we propose DREAM, a DistRibuted Energy-Aware traffic Management framework for data center networks, with improved responsiveness and better energy savings than current centralized approaches [49, 113]. In DREAM, the distributed agents and hosts work cooperatively to consolidate traffic to a portion (e.g., the left-most) of the DCN. It reacts to traffic bursts at the Round Trip Time (RTT) time scale. The

network monitoring is based on data plane of the hardware switch rather obtaining values of flow counters of the forwarding plane through Openflow [73]. The distributed agents are implemented as a shim layer, such as Open vSwitch [89]. DREAM requires no modification to the host protocol stack or the data center networking fabric. This makes our design easy for deployment. Testbed evaluations using traces from Wikipedia web service [107] and Facebook MapReduce traffic [19] prove that DREAM on average achieves at least 15.8% energy saving for the data center network, while state-of-the-art approaches such as CARPO [113] and ElasticTree [49] produce 11.6% and 8.4% energy saving, respectively. The packet drop ratio in DREAM is less than 0.01% while the best among the alternatives, ElasticTree, has at least 0.19% drop ratio. DREAM also has 30% lower application-level latency than state-of-the-art centralized traffic consolidation approaches.

Contribution 3: Joint Server and Network Energy Optimization. Latency critical applications in data centers follow a request/response model and usually exhibit "ON/OFF" patterns [24]. Exploiting the latency slack of a request by slowing down its processing is another major approach for saving energy in data centers. In this approach, DVFS based frameworks such as Rubik [60] and Pegasus [69] run the CPU cores at the minimum frequency that satisfies the request's target deadline. Other sleep states based schemes such as DynSleep [24] and SleepScale [68] postpone the servicing of requests and cause a longer idle period so that servers can enter into their deepest sleep states. Nevertheless, prior DVFS or sleep states techniques [60, 69, 24, 68] make servers energy proportional without considering power managements in the DCN. Traffic consolidation policies in the DCN will affect the amount of network slack that can be used by server power manage-

ment schemes. Power savings that are achieved by only optimizing the server slowing-down technique or DCN traffic consolidation is considerably less than jointly solving both the problems.

Such findings motivate a joint server and network energy saving framework for latency-critical applications named EPRONS (i.e., Energy PROportional Network and Server). In EPRONS, we minimize the overall data center's power consumption with latency-sensitive applications by trading-off network slack in favor of providing additional slack for computations. We utilize the linear programming model to consolidate latency-sensitive search queries and latency-tolerant background flows to a minimal subnet of the topology by turning off unused switches and links without violating the application deadlines. Servers take advantage of the additional 'network-provided' slack to allow slowing down request processing. For servers, we design a novel power saving technique using DVFS based on the *average tail latency* of a request. If needed, we turn on a minimal number of additional network links and switches to reduce network latency while still maximizing entire data center's power saving. Experimental results show that our scheme saves up to 31.25% of a data center's total power budget.

Contribution 4: Neural Network based Two-Step DVFS Scheme. Web search is one of most important latency-critical applications in data centers, which is critical to the service provider's revenue [12]. The service quality of web search is significantly affected by the request processing latency at Index Serving Nodes (ISNs) [72], which store the document index and retrieve matching results for a query. Power managements for ISNs seek to estimate each request's service time and then properly re-configure the current

frequency scaling or sleep states setup. A common approach for service time prediction [69, 108, 75, 23, 60] is to assume that each request's CPU cycles can be estimated from the same single distribution, based on offline profiling. By using the high percentile tail of this service time distribution and then deriving the CPU frequency or sleep states, they seek to achieve a low deadline violation rate while also slowing down search queries. However, search requests' latencies have both short and long term variation and a request's computation requirement (i.e., total CPU cycles) cannot be predicted accurately [60, 48]. This results in either energy inefficiency or even worse, deadline violations. Hence, maintaining a balance between energy savings while meeting deadlines on the ISN is necessary.

In this dissertation, we present Gemini, a novel power management framework for latency-critical search engines. Gemini has two unique features to capture the per query service time variation. First, at light loads without request queuing, a two-step DVFS is used to manage the CPU power. Our two-step DVFS selects the initial CPU frequency based on the query specific service time prediction and then judiciously boosts the initial frequency at the right time to catch-up to the deadline. The determination of boosting time further relies on estimating the error in the prediction of individual query's service time. At high loads, where there is request queuing, only the current request being executed and the critical request in the queue adopt a two-step DVFS. All the other requests in-between use the same frequency to reduce the frequency transition overhead. Second, we develop two separate neural network models, one for predicting the service time and the other for the error in the prediction. The combination of these two predictors significantly improves the power saving and tail latency results of our two-step DVFS. Gemini is implemented on

the Solr search engine. Evaluations on three representative query traces show that Gemini saves 41% of CPU power, and is better than other state-of-the-art techniques.

Contribution 5: Coordinated Time Budget Assignment at the Aggregator. Another important topic for power management in web search is determining each request's time budget [123, 53, 22], as it directly affects the available latency slack on ISNs and thus the resulting power savings. In general, a request's time budget is assigned by the aggregator of distributed search engines [16, 11], and is broadcast with search query to all the participating ISNs. With exhaustive search, the aggregator has to wait for all ISN responses and return the top-K ranked results to the client [81]. This wastes system resources [97, 65] and significantly degrades a search engine's response time. To reduce the tail latency, aggregation polices [123, 53, 22] such as FSL [123] rank the ISNs based on the query's response times and optimize the ISN cutoff parameters for a set of queries. They assign the request's time budget by assuming a stable request pattern during a short time period and consider all the ISN responses during the period irrespective of their quality contributions. What is worse, requests that can not be finished within their given time budgets are still executed on ISNs, which waste server power. Selective search [104, 65, 97, 81], on the other hand, improves the epoch-based aggregation polices with a query-specific ISN cutoff based on its quality contribution. It leverages the individual query's information and a Central Sample Index (CSI) [97] at the aggregator to rank ISNs. However, it is difficult to make an accurate prediction of the quality contribution based on a sample index, hence it often results in a second cutoff prediction [81]. A large CSI produces a better quality prediction, but incurs large additional delay for the search at the aggregator. What is needed

is a coordinated decision between the ISNs and the aggregator to take advantage of local predictions at the ISN combined with the overall time budget and ISN cut-off decision by the aggregator.

This dissertation proposes Cottage (i.e., **co**ordinated **t**ime budge**t** **a**ssi**g**nm**e**nt), a coordinated framework between the aggregator and ISNs for latency and quality optimization in web search. Cottage has two separate neural network models at each ISN to predict the quality contribution and latency, respectively. Then, these prediction results are sent back to the aggregator for latency and quality optimizations. The key task is integration of the predictions at the aggregator in determining an optimal dynamic time budget for identifying slow and low quality ISNs to improve latency and search efficiency. Cottage accelerates slow ISNs with a high quality contribution for improved search quality. Our experiments on the Solr search engine prove that Cottage can reduce the average query latency by 54% while searching fewer ISNs and documents than state-of-the-art techniques. It is shown that Cottage can still achieve a good P@10 search quality of 0.947, which is much higher than the other techniques.

The rest of this dissertation is organized as follows. Chapter 2 summarizes the related works in different categories. Then, we introduce the graph-based container placement in chapter 3. Chapter 4 presents our distributed traffic consolidation. In Chapter 5, we propose the joint server and network energy optimization for latency-sensitive applications. Chapter 6 focuses on the neural network based two-step DVFS on ISNs, followed by the aggregator design of coordinated time budget assignment in chapter 7. Finally, chapter 8 concludes this dissertation.

Chapter 2

Related Work

This chapter covers the literature related to the studies proposed in this thesis.

2.1 Resource Provisioning in Data Centers

There have been several studies [111, 27, 117, 30] focusing on the resource scheduling in data centers to improve task completion time or energy efficiency. Most of them are based on two techniques: the static VM placement [111, 27] or dynamic VM migration [117, 30]. E-PVM [8] places VMs on a server with the lowest utilization to leave a large headroom for load spikes and has good task completion time. Google's Borg [111] achieves a high cluster utilization by using the priority-based task scheduling, task preemption and task packing. To increase the packing efficiency, Borg aims to reduce stranded resources [111] when only some but not all resources on a machine are fully allocated. RC-Informed [27], on the other hand, finds that VM behaviours are consistent over multiple lifetimes. It designs a system that collects VM's statistics offline and predicts its future resource demands

online. The VM scheduler in RC-Informed divides a physical machine into multiple slots and over-subscribes each VM slot's CPU resources. Instead of increasing the packing efficiency, the mPP algorithm in pMapper [112] places VMs on servers having the lowest power increase per unit of utilization. The above approaches improve either power consumption [111, 27, 112] or task completion time [8], but are unable to achieve improvements in both dimensions. Besides server resource constraints, some other related works [127, 13, 120] also factor the network bandwidth constraint when packing VMs. PowerNetS [127] puts the source and destination VMs of a network flow together to improve the network latency as well as server energy efficiency. Oktopus [13] considers the network bandwidth as the primary constraint in resource allocation for tenants in the cloud. Subsequently, Proteus [120] extends that work by adding time variability in network bandwidth reservation.

Live VM migration [26] is another important technique to improve the resource utilization of data centers, whiling meeting performance constraints. Cloudward [46] designs a sophisticated migration strategy for populating a service from an on-premise data center to the public cloud. It aims to find the optimal partition for enterprise operations such that important performance metrics are maximized. eTransform [99] is another VM migration framework for running enterprise infrastructure among different data centers. The linear programming model in eTransform tries to optimize the operational cost of services across multiple data centers. Cloudnet [117] and VMShadow [44] are pre-copy migration techniques, seeking to migrate application state while trying to minimize application downtime. In Remus [30], the post-copy technique is used for VM replication in failure recovery. Post-copy only migrates the minimal working set of VMs and the incoming requests are

transfered to the target server immediately. Container migration has been discussed recently. It is challenging to manage the process tree checkpointing and restore. All of the existing works use the CRIU project [28] for container migration. KUP [59], proposes to use process checkpoint & restore for fast kernel updates. Several optimizations, such as incremental checkpoint and on-demand restore are also implemented in KUP. Container checkpoint & restore has also been used in the public cloud. Picocenter [124] checkpoints only the long-lived user containers that are most-idle, checkpointing them into persistent storage such as Amazon S3. These container images are restored when a packet arrives for the application.

2.2 Traffic Consolidation in the DCN

Gupta et al. [45] first proposes a position paper exploring possible opportunities to save Internet power. At the device level, idle components of a switch can be put into sleep. At the network level, traffic can be rerouted to a few links while letting other idle switches go to sleep. The following works [2] and [84] demonstrate that DCN can be energy-proportional at the device level. GoogleP [2] combines the flattened butterfly topology and rate adaption technique to dynamically change link speed and power consumption. Research [84] buffers the network packets at edge switches for a little while to have longer traffic gaps. In such a way, intermediate switches can have longer sleep time.

ElasticTree [49] proposes three centralized optimizers which consolidate traffic to the minimal subnet of the network and turn off the unused switches. Traffic consolidation in ElasticTree can have better power saving than device level frameworks. However, it is

impractical in production data centers because of its high computation complexity. To improve the scalability, GreenTE [125] proposes a heuristic algorithm to accelerate centralized traffic consolidation. The distributed design is a promising approach to make the traffic consolidation more responsive to traffic variation in the DCN. EATe [110] solves a complex linear programming model at edge switches to consolidate traffic on pre-defined paths. Similarly, DISCO [128] proposes the host level and link level distributed traffic consolidations. But they still use the centralized controller to implement the distributed designs, modeled as a number of sub-optimizers. Another distributed design, REsPoNse [109], requires the full traffic history as prior knowledge to train their algorithm to derive the always-on path offline, over which traffic is routed. But this depends on the assumption that the data center traffic pattern doesn't change significantly, which is not the case in real DCNs [14]. REsPoNse [109] relies on proactive link utilization feedback from routers to perform traffic scheduling, which is also difficult in current hardware switches. So, REsPoNse is only implemented in the CLICK software switch [64]. CARPO [113] considers traffic correlation while consolidating traffic. However, it still uses the complex centralized optimizer. Finally, FCTcon [126] designs the deadline aware traffic consolidation in the DCN with network latency feedbacks.

2.3 Power Management with Latency Constraints

To improve the energy efficiency of data center servers, most of the existing work on power management is based on DVFS or Sleep states techniques. PowerNap [74] dynamically switches the server state between a minimal power consumption "nap" state and

a high performance active state, to accommodate workload variations. Based on PowerNap, DreamWeaver [75] coalesces requests across multiple cores so that some cores can enter deeper sleep states. Instead of using request coalescing, DynSleep [24] procrastinates the processing of requests at a single core while still avoiding deadline violations. By doing this, the CPU core can enter the deepest sleep states to save more energy. KnightShift [115] designs a heterogeneous server architecture by having a low power consumption "Knight" node. When the server utilization is low, KnightShift puts the entire server into sleep states, except for the Knight node. The low utilization workloads are served by the Knight node when other server components go to sleep. SleepScale [68] provides a method to combine different power management frameworks. On the other hand, Pegasus [69] proposes a feedback based DVFS scheme to save server power. In Pegasus, it measures the request's latency periodically and selects the highest CPU frequency if a deadline violation happens. When the measured latency is smaller than 65% of given time budget, the CPU frequency is reduced. Similarly, TimeTrader [108] considers both network slack and server slack to save power for latency-critical applications. When the ECN or RTO signals are not presented in the DCN, its power management framework will add the full network latency budget to the compute slack.

Rubik [60] observes that both the request service time and inter-arrival time have high short-term variability. To capture the per request variability, Rubik develops an analytical model for CPU frequency selection at the granularity of every request arrival and departure. However, it only approximately estimates each request's computation demand, as every query's CPU cycle need is derived from the same distribution. For latency con-

straints, it conservatively uses the tail of service time distribution for each request's latency prediction, thereby potentially missing many energy saving opportunities. Similar to Rubik, μDPM [23] proposes a power management scheme for applications with microsecond service times. Its analytical model combines the request delaying, sleep states and frequency scaling to save power on CPU cores. A few researchers have theoretically shown that a step-wise DVFS [70] can produce better energy savings even though the actual service time is unknown [70, 118]. While the two previous papers dealt with minimizing the energy consumption of a single request, the work in [48] addresses multiple arrivals based on adjustments at a time epoch. All the above works [70, 48] utilize sampling of the distributions to estimate the residual task time and then select the next step in the CPU frequency. The profiled distributions have to be obtained offline as a prior knowledge. Another important feature of some prior research [48, 118, 47] is that they need feedback signals to make the control decision, with the same criteria to enter the next step of frequency.

2.4 Latency and Efficiency Optimization in Web Search

Aggregation policy [22, 123, 55, 53] of web search determines how long it waits for the ISNs' responses in attempting to reduce overall latency. FSL [123] categorizes the ISN's responses into three categorizes and only optimizes the search latency by cutting off the stragglers, leaving the fast and long queries untouched. Its design utilizes an epoch-based offline algorithm for threshold determination in order to have an acceptable search quality for most of the queries. Unlike cutting off the responses but still executing them, TailCut [22] drops the search request at the ISN to improve efficiency. It is still an epoch based

design such that a second query has to be sent to all the ISNs whenever the measured search quality is below a threshold.

Selective search [97, 104, 65, 62, 81] estimates the relevance of each ISN for a specific query and only chooses the most promising ISNs to search. ReDDE [97] has a CSI at the aggregator. On every request arrival, it is first looked up at the CSI to get the top-K results. Then, ISNs are ranked according to their contributions to the top-K sample results. Rank-S [65] also uses the CSI but ranks the ISNs in a different approach. Specifically, they first utilize the matched documents from the CSI to form a tree structure. The matched documents for a given query are leaves of the tree from left to right in descending order of scores [65]. Every leaf node's score is normalized by considering the original score as well as its distance to the left-most leaf node in the tree. An ISN's final quality estimation is the summation of all its documents' normalized scores. Kim et al. [62] proposes a shard ranking algorithm considering each ISN server's system load. While all these works focus on the ISN ranking and have a fixed ISN dropping threshold, QR [81] develops a machine learning based model to predict the cutoff assuming the existence of a perfect ISN ranking. Additionally, it extends the resource selection algorithm to the search type of recall-driven. One major drawback of the previous CSI based frameworks is their poor scalability. Taily [7] proposes a distributed approach that assumes the score distribution on an ISN follows a Gamma distribution [58]. It predicts the parameters of the Gamma distribution at runtime based on static query term statistics. Although Taily improves the scalability, we find that a query's scores typically does not perfectly fit a Gamma distribution, thus resulting in inaccurate ISN cutoffs. Also, the local prediction of quality at the ISN does not reflect

the overall quality at the aggregator, which is based on the combined ranking of all the responses from the ISNs.

Chapter 3

Goldilocks: Graph based Container Placement

3.1 Introduction

This chapter presents a graph-based approach, Goldilocks, to elegantly solve the complex resource 'right sizing' problem in data centers to optimize both power and task completion time. In Goldilocks, two kinds of graphs are considered in the partitioning algorithm: a) a capacity graph, representing data center's total resources and b) a container graph with resource demands as the vertex weights and inter-container communication as edge weights. By running the recursive graph bipartitioning algorithm in METIS [77] with the min-cut objective function, containers with high communication frequency are grouped together. The load for each container group is automatically balanced in the algorithm. Goldilocks significantly improves the task completion time as containers with

frequent communication are placed close together in the DCN topology. The power saving is achieved by packing groups of containers into a minimal number of servers, so that unused servers are turned off. We first solve the container group packing problem in symmetric Clos topology. We then extend our algorithm to the asymmetric topology with heterogeneous servers.

In addition to the graph partitioning algorithm, we re-examine the conventional wisdom regarding power management [74, 49, 60]. A common sense strategy for energy efficient data centers has been to run some servers close to 100% utilization (with a small safety margin [49]), and turn off idle servers for maximal power savings [35, 96]. However, SLA violations for latency-sensitive applications [91] can be a major concern when servers become over-loaded due to burstiness of the workload. In Goldilocks, we operate servers at *Peak Energy Efficiency* (60-80% utilization) [116], which is defined as the point to achieve the maximum number of operations completed per watt. Such a strategy saves more total server power and leaves a much larger headroom to deal with instantaneous load fluctuations.

To demonstrate the feasibility of Goldilocks, we implemented a seamless Docker container migration framework on a 16-server testbed. For the Twitter content caching application, Goldilocks saves 11.7% power, with the variation seen in Azure cloud trace [27] and 22.7% power on the variability seen in Wikipedia trace [107]. Comparatively, the best of alternatives (i.e., Borg) saves 8.9% and 21%, respectively. Although Borg and Goldilocks have somewhat similar power saving results, the energy consumption per request for Borg is 3.5 times that of Goldilocks. Because of the locality-focused partitioning and

large headroom for load fluctuation, Goldilocks produces at least 2.56 times of better task completion times, compared with any of the implemented alternatives. Our large scale trace driven simulation, based on Microsoft traces [6], reiterates the significant power saving and reduced task completion time of Goldilocks. Our major contributions include:

- We propose a holistic graph-based approach for resource provisioning in containerized data centers. This approach places containers closely to minimize power consumption and task completion time.
- A number of resources and performance considerations are taken into account, such as CPU, memory, network, migration overheads and workload fluctuation.
- We propose to achieve *Peak Energy Efficiency* for servers that maximizes performance for the energy consumed.
- We implement Goldilocks in a data center testbed, including seamless Docker container migration between servers. We show results both from the implementation and a large scale simulation based on a Microsoft Azure trace.

3.2 Peak Energy Efficiency

In energy efficient computing, the basic assumption is that server power increases linearly as we increase the utilization [115, 68]. This linear power curve assumption is the basis for a number of efforts [67, 91, 69] to consolidate server load, turn off idle servers and thus save static server power. Usually, the servers are packed to 100% maximal utilization. However, the latency of application might increase due to longer queuing time. The SLA

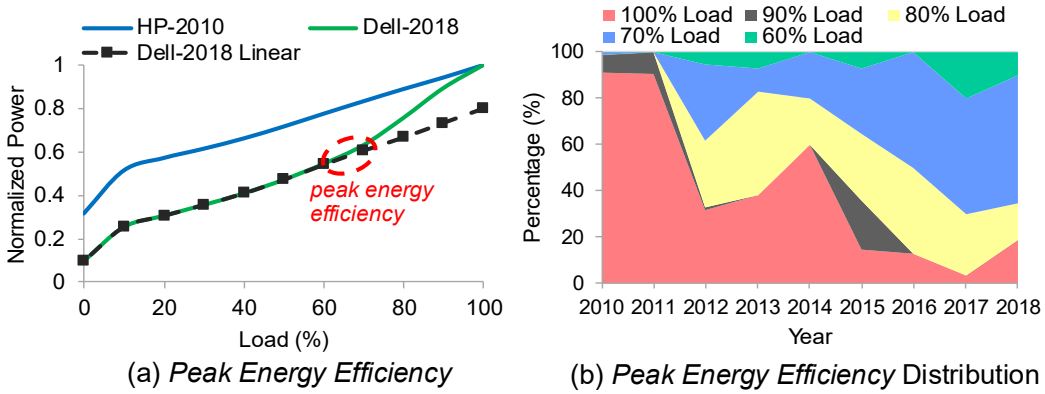


Figure 3.1: The distribution of Peak Energy Efficiency utilization for the SPEC power benchmark.

violation becomes a concern as there is little room to accommodate workload variation [108].

We observed that the server power increases quickly beyond a point on the load vs. power curve. Load means the quotient of current request rate and maximum possible request rate on the server. In this chapter, we define *Peak Energy Efficiency* as achieving the maximum number of operations per watt [116]. When servers operate at higher utilization than the *Peak Energy Efficiency*, power increases faster than the load because of automatic frequency boosting [130] and increase in temperature requiring higher fan speeds [91]. Fig. 3.1 (a) shows the normalized power of 2 recent servers, as we vary the load. The power is normalized to maximum power consumption at 100% load, to avoid focusing on the differences among systems from different vendors. The major take-away of this result is that the power consumption vs. load was mostly linear until 2010, but not any more. As also observed with the Dell-2018 curve, power increases much faster beyond the *Peak Energy Efficiency* point [116]. For comparison, the dotted line shows what would be a linear increase if it was strictly power proportional.

The cubic power curve after *Peak Energy Efficiency* utilization in Fig. 3.1 (a) is due to the DVFS [69, 115, 130, 60] on modern server. In DVFS, power P equals $C*V^2*f$, where C is capacitance, V is voltage and f is frequency. At low loads, the voltage V does not scale down further because it is already low. Only the frequency f scales down. This explains the linear power curve below the *Peak Energy Efficiency*. At high loads, both voltage V and frequency f will scale up, which results in the increase in power according to the cubic law. Fig. 3.1 (b) shows the *Peak Energy Efficiency* utilization results of 419 servers subjected to the SPEC request/response transaction workloads [103]. The *Peak Energy Efficiency* utilization for each server is obtained by analyzing the SPEC power benchmark results uploaded by vendors. This benchmark exercises the CPU, Cache, Memory, I/O as well as the operating system. 100% load is defined as the maximum number of requests that can be supported by the server. The Y-axis on Fig. 3.1 (b) is the share of each listed *Peak Energy Efficiency* utilization (i.e., 100% to 60% load) for a specific year. Each color represents one kind of *Peak Energy Efficiency* utilization. We can observe that most of the servers in 2010 have the *Peak Energy Efficiency* at 100% server utilization. During recent years, the *Peak Energy Efficiency* utilization of servers has already moved to the range of 60-80% utilization.

Let us examine how running servers at *Peak Energy Efficiency* can be beneficial in a data center context. There are two benefits: better power saving and higher tolerance for workload fluctuation. Consider placing a group of containers in a cluster of 1000 servers. Fig. 3.2 (a) shows that fewer servers are needed as we increase the load per server. The corresponding total power consumption is depicted in Fig. 3.2 (b). Servers have the same

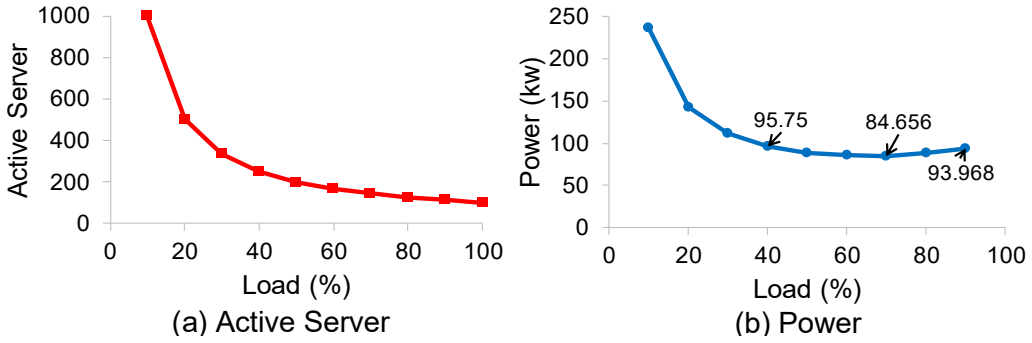


Figure 3.2: Operating servers at the Peak Energy Efficiency utilization consumes the least total server power.

power model as Dell-2018 in Fig. 3.1 (a), which has *Peak Energy Efficiency* at 70% utilization. We can observe an obvious 'U' curve for total server power. The maximum power is saved when servers are packed only up to 70% utilization.

Lowering the utilization allows us to tolerate burstiness of data center workloads [14, 93, 27] and have shorter latency. Workloads across applications in a cloud data center might also be correlated [25]. We calculated the Pearson correlation of 1500 VMs in the Microsoft Azure trace [27]. 99.8% of time the Pearson correlation is between 0.6 and 0.8, indicating (pair-wise) that VMs might 'burst' at the same time. Packing servers to the *Peak Energy Efficiency* leaves sufficient headroom to accommodate burstiness in the workload. Using just bin packing with a target of 100% server utilization [67, 91], tasks have to be migrated when the server becomes overloaded to avoid violating the application's SLA requirement.

In the DCN, we turn off idle switches and links. Table 3.1 gives the configuration of 3 different data centers such as Google's Jupiter [98] and Microsoft's VL2 [43]. Because the power models are not given [98, 39, 43], we carefully select power models from the

Table 3.1: The configuration of 3 different data centers.

	# of Server	# of Switch	# of Link	Power Model
Google [98]	98304	2048 ToR 3584 Fabric	147456	96W SoC server (Facebook 1S [85]) 630W ToR/Fabric switch (2 HPE Altoline 6940 [50])
Facebook [39]	184320	4608 ToR 576 Fabric	36864	96W SoC server (Facebook 1S), 282W ToR (Facebook Wedge [86]), 1400W Fabric (Facebook 6Pack [86])
VL2(96) [43]	46080	2304 ToR 144 Fabric	9216	250W Microsoft blade server [85], 282W ToR (Facebook Wedge), 1400W Fabric (Facebook 6 Pack)

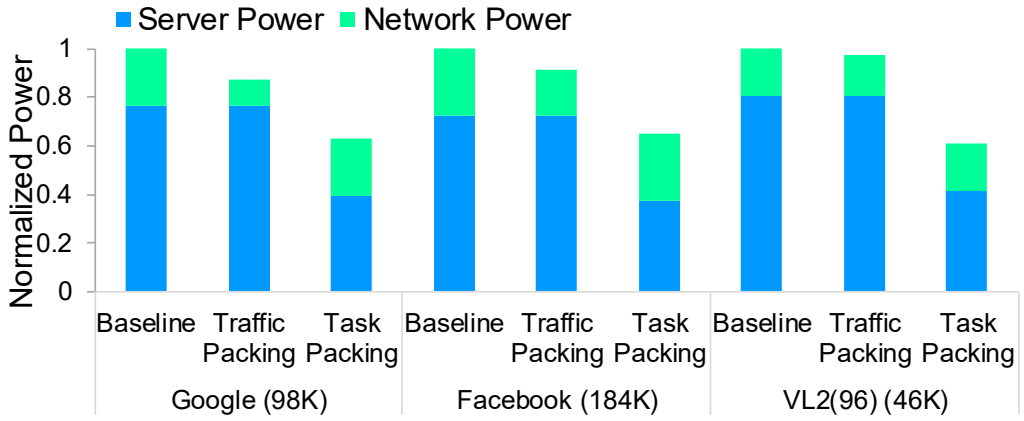


Figure 3.3: The power breakdowns on 3 different data centers.

Open Compute Project [85, 86] to match the switch's port density and server's network bandwidth. For example, the Facebook's DCN topology has 10G servers, 16X40G Top of Rack (ToR) switch and 96X40G Fabric switch. So, they are mapped to Facebook 1S System on Chip (SoC) server [85], Facebook Wedge ToR switch and Facebook 6 Pack Fabric switch [86] accordingly.

Power breakdowns for the 3 different data centers are depicted in Fig. 3.3. At the baseline, all the servers are uniformly loaded at 20% utilization [74] and link utilization between ToR switch and next stage switch in the Clos topology [3] is 10% [93]. To ease the

comparison, all the power results are normalized to baseline. The first take-away in Fig. 3.3 is that DCN only contributes around 20% of the total power for all the 3 data centers. The results are in line with prior works [49, 109]. Next, we focus on the power saving results of Traffic Packing in the network and Task Packing on servers. By saying Traffic Packing, we mean moving all traffic to the fewest number of links and switches as long as they are not overloaded. In Task Packing on servers, all the loads are packed into the fewest servers as long as the total utilization of the server is below a threshold. The results are obtained through mathematical analysis of bin packing [122]. The second take-away is that Traffic Packing on average can only save 8% of the entire data center's power, while Task Packing saves as much as 53% of total power. Thus, we should better do the Task Packing on servers to save most of the power.

3.3 Provisioning on Symmetric Topology

In prior works, each container is allocated to a server independently, ignoring the affinity or dependency between containers. Goldilocks leverages a holistic graph-based approach to solve the container placement problem, while minimizing power *and* task completion time. The algorithm is centered around the partitioning of two graphs: the capacity graph and container graph. Goldilocks partitions the containers into different balanced groups before assigning each container group to a subtree in the topology. The grouping is achieved by running the recursive bipartitioning algorithm on the container graph. With the min-cut objective function, the containers having high communication frequency are grouped together and then placed closer together in the data center.

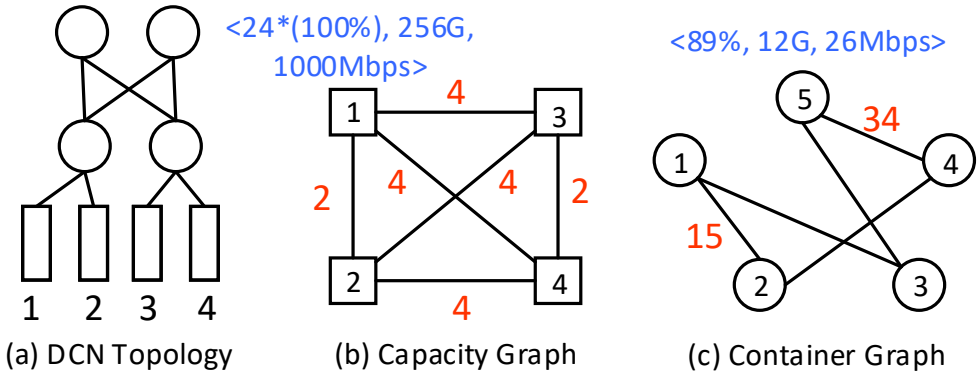


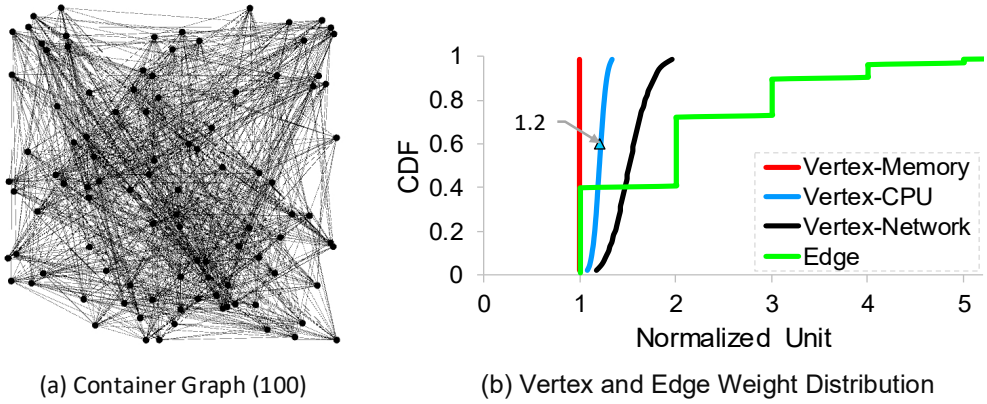
Figure 3.4: Example for capacity graph and container graph.

3.3.1 Graph Construction

Capacity Graph: We construct a capacity graph, representing the total resource capacity in the data center. The vertex weight is a 3 dimensional vector <*CPU utilization, Memory usage, Network bandwidth*>. The edge weight is the length of the shortest path (i.e., number of links) between server pairs in the topology. As an example in Fig. 3.4 (a), we show a simple topology with 4 switches (circles) and 4 servers (rectangles). Its capacity graph is shown in Fig. 3.4 (b). The vector <24*(100%), 256G, 1000Mbps> means that each server has 24 CPU cores (with 100% utilization), 256GB memory and 1000Mbps network bandwidth (we choose to not factor in the disk size for the moment, assuming it is not likely to be a limiting factor). With the Clos topology allowing line rate communication between any server pair [3], we set the *Network bandwidth* in vertex weight as the link capacity of server's Network Interface Card (NIC). We relax this assumption in section 3.4. We set the edge weight as path length, because each substructure in the Clos topology will automatically be grouped together during the graph partitioning for the max-cut, since inter-substructure edges always have the largest edge weight.

Table 3.2: Vertex weight and edge weight of 4 data center workloads.

	CPU (%)	Memory (GB)	Network (Mbps)	Flow Count
Twitter Content Caching (Memcached)	33	4	24	4944
Web Search (Apache Solr)	32	12	1	50
Naive Bayes Classifier (Hadoop)	376	2	328	2
Media Streaming (Nginx)	54	57	320	25



(a) Container Graph (100)

(b) Vertex and Edge Weight Distribution

Figure 3.5: Vertex weight and edge weight distribution (part b) in the Microsoft search trace graph (part a, 100 vertices snapshot).

Container Graph: Each vertex in the container graph corresponds to a container in the workload. The vertex weight is the resource demand, in terms of CPU utilization, memory usage and network bandwidth. The edge weight is the number of distinct flows between the pair of containers. We aim to place together the containers with the most frequent communication. An example of the container graph is given in Fig. 3.4 (c).

We present the characteristics of different containerized applications, deployed in our testbed, in Table 3.2. Each application instance is hosted in a Docker container. Columns 2-4 are the vertex weights and the last column is the edge weight. Next, we show a part of the container graph for the Microsoft search trace [6] (which has 5488 vertices

and 128538 edges) in Fig. 3.5 (a) for 100 vertices (IP range: 10.0.0.1 to 10.0.0.100 in the trace). The vertex weight and edge weight distribution are plotted in Fig. 3.5 (b). All the weights are normalized to the smallest value in the distribution. For example, the dot on the Vertex-CPU line with the value of 1.2 on the X-axis means its CPU utilization is 1.2 times larger than the smallest CPU utilization in the trace. In the trace, all the nodes performing the search indexing occupy 12GB memory for in-memory index accessing. So, the vertex weight is 1 for memory usage, after normalization for all vertices.

3.3.2 Container Partitioning and Assignment

Goldilocks assigns a group of containers that have inter-dependencies together, rather than assigning each stand-alone container. The group is assigned to a substructure (a machine, a rack, a pod or a subtree) in the DCN topology. The substructure can be automatically found by recursively bipartitioning the capacity graph, using the max-cut objective function. The resource capacities of every server in the substructure are factored as vertex weights in capacity graph. The group of containers are assigned to a substructure only if the containers' resource demands can be satisfied by the substructure.

Consider a container graph $G^c = (V^c, E^c)$, where V^c are the vertices and E^c are the edges. The number of vertices is given by $m = |V^c|$. A^c is the vector representing the resource demands of container vertex V^c. Similarly, $G^t = (V^t, E^t)$ is the capacity graph with N vertices . B^t is the vector representing the resource capacity of server V^t. The goal of partitioning on the container graph is to find n partitions P_1 to P_n such that:

$$minimize \sum_{1 \le i < j \le n} |E_{ij}^c| \tag{3.1}$$

$$\forall\ P_i, \sum_{\forall j \in Pi} A_j^c \le B_i^t, where\ 1 \le i \le n \tag{3.2}$$

$$U_{P_1} \approx U_{P_2} \approx ... \approx U_{P_n} \tag{3.3}$$

n is determined at run-time of the partitioning algorithm rather than as a pre-defined static parameter. U_{P_i} refers the utilization of server V_i^t, where container group P_i is hosted. In equation (2), the algorithm stops bipartitioning the container graphs until the container group's resource demands can be satisfied by the server's resource capacity. At the final step of the partitioning algorithm, the container groups are assigned to servers. Equation (3) guarantees that the containers are uniformly distributed among n partitions. E_{ij}^c is the set of edges between partition P_i and P_j in container graph G^c. $|E_{ij}^c|$ refers the sum of edge weights for E_{ij}^c. Equation (1) guarantees that the cut is minimized when partitioning the container graph. Equation (2) guarantees the resource constraints are satisfied.

Fig. 3.6 shows the workflow for locality partitioning. G_0 in Fig. 3.6 (a) is the initial container graph and Fig. 3.6 (b) is the DCN topology. G_0 is partitioned into G_{11} and G_{12}. In the next iteration, G_{11} and G_{12} are bipartitioned because their resource demands exceed the server's resource capacity. Because the graph partitioning algorithm in open-source software, such as METIS [77], can tolerate some imbalances between container partitions, the number of vertices or total resource demands in G_0 does not need to be the power of 2. For example, the partition G_{23} can not be assigned to a server without violating resource constraints. But, its counter-part G_{24} is a little smaller than G_{23}, as the parent partition G_{12} cannot be ideally partitioned into two equally balanced partitions. Thus G_{24} can be

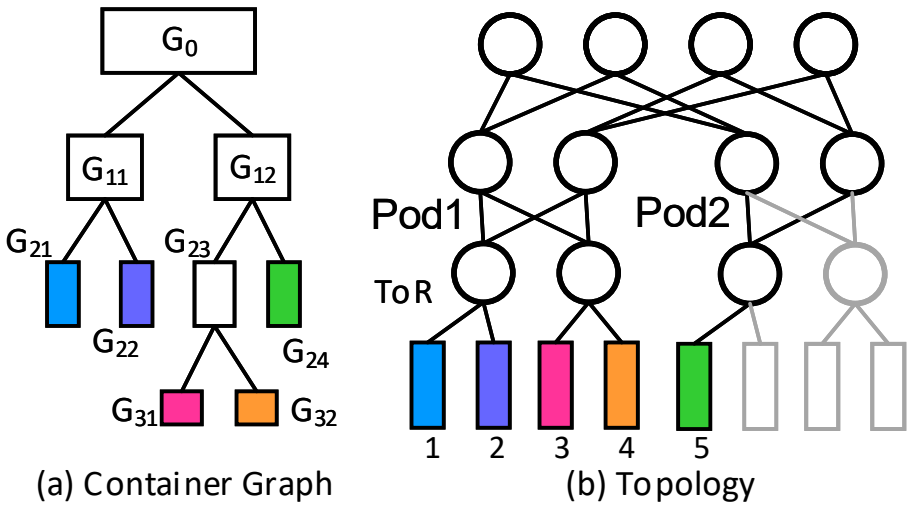
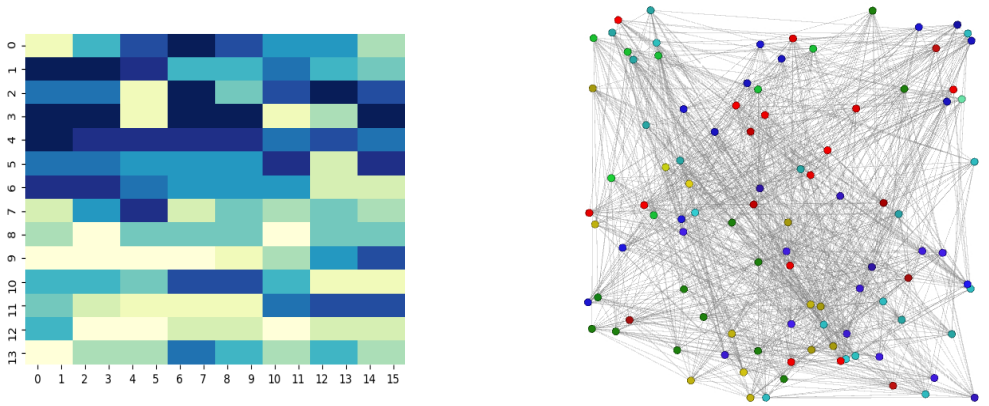


Figure 3.6: Recursively bipartition the container graph until the resource demands of leaf nodes (in part a) can be satisfied by the resource capacity of servers (in part b).

successfully allocated to a server. The partition G_{23} has to be bipartitioned again to obtain G_{31} and G_{32}.

Finally, the container groups to be assigned to the servers in Fig. 3.6 (b) are G_{21}, G_{22}, G_{31}, G_{32}, G_{24}. In addition to maximizing the intra-group locality, the partitions G_{21} and G_{22} that have the same parent partition are also assigned to the same rack in Fig. 3.6 (b) to maximize inter-group locality as well. By using the bipartitioning algorithm recursively, the inter-container communication is localized to a server, a rack, a pod or a subtree in the topology.

The partitioning algorithm in Goldilocks is implemented by using METIS [77]. METIS produces the optimal balanced min-cut partitioning. The computation time is reasonably small. For example, it just takes 285s to partition a graph with 1 million vertices. That means the epoch length in Goldilocks from one execution of container placement to the next can be short enough to quickly adapt to load changes in data centers. It is practical



(a) Twitter Content Caching (b) Microsoft Search Trace

Figure 3.7: Partition results for the Twitter Content Caching with 224 containers (part a) and Microsoft search trace with 100 vertices (part b).

to deploy the algorithm for a large topology with millions of containers. In Fig. 3.7, we present two real partitioning results from our testbed experiment and also in the Microsoft trace graph. In Fig. 3.7 (a), there are a total of 224 Memcached containers (to simulate the Twitter content caching) which are represented by a cell. Each color in the results means a unique partition. Similarly, for the Microsoft trace graph [6] in Fig. 3.5 (a), there are 5 different container partitions shown in Fig. 3.7 (b).

3.4 Provisioning on Asymmetric Topology

In section 3.3.2, we assumed a symmetric network topology with full bisection bandwidth and homogeneous servers. Although symmetric topology is widely used [98, 93, 3, 49, 130, 113, 43, 122], switch and link failures can make the DCN topology asymmetric. Thus, servers are not inter-changeable because of imbalanced network bandwidth in various parts of the DCN [5]. The homogeneous server assumption might not hold as well [111, 31] because of legacy equipment even in a data center with custom-built servers [98].

In order to relax the symmetric topology and homogeneous server assumptions, we have to overcome two problems. First, only checking the network bandwidth usage at the server's NIC is not enough to ensure that resource demands are met, as the full bi-section bandwidth assumption doesn't hold with an asymmetric DCN topology. Second, Goldilocks partitions the containers into a few balanced groups and maps them to homogeneous servers in section 3.3.2. But now, each server has different computing and networking capabilities.

We now solve the problem of allocating m containers to an asymmetric tree topology without violating the resource constraints for each server and network link. The servers have various CPU cores, memory capacity and NIC bandwidth. Containers are labeled with a particular Group id for the sake of minimal power consumption and task completion time. Containers with same Group id should be placed close to each other in the topology. This is a NP-hard problem [37, 69]. So, we propose a heuristic algorithm in Goldilocks. In section 3.4.1, the algorithm is first analyzed under the assumption that there is no inter-group communications between containers. Subsequently, we consider the more realistic case that container groups communicate with each other.

3.4.1 Assignment without Inter-Group Communication

The heuristic algorithm reuses the locality-focused partitioning in section 3.3.2. The bipartitioning algorithm stops when the resource demands of every container group can be satisfied by the average capacity of the heterogeneous servers. If m containers are partitioned into n groups, the initial number of active servers to place the m containers is n. Because the average capacity is used, the final number of active servers n' might be

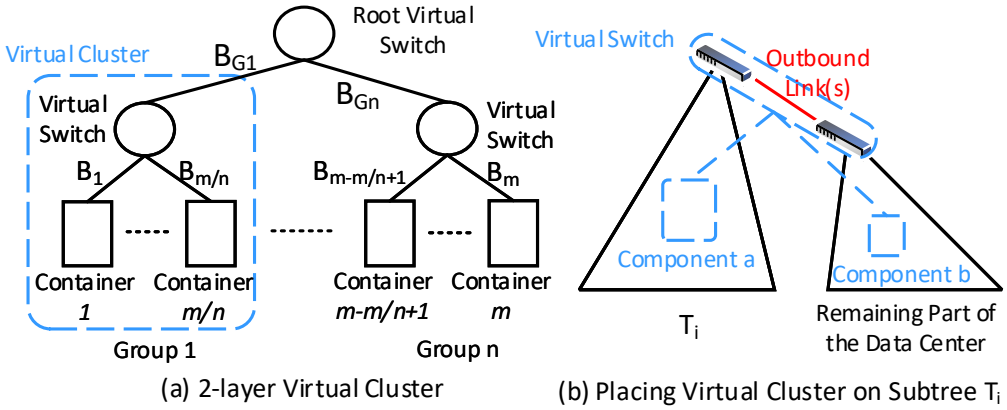


Figure 3.8: In (a), each container group is abstracted as a Virtual Cluster. All the running containers are represented by a 2-layer Virtual Cluster. When placing a virtual cluster of containers on the DCN topology, any underlay physical link divides the virtual cluster into 2 components (part b).

larger than the initial value n after allocating all the m containers. The abstraction of a Virtual Cluster, introduced in Oktopus [13] and Proteus [120] is leveraged by Goldilocks to represent a container group.

In Fig. 3.8 (a), a total of m containers are partitioned into n groups. Containers 1 to m/n are grouped together into a Virtual Cluster, and these containers are connected by a virtual switch. The virtual switch would represent a group of physical switches and links of the DCN, and it is expected that it can support the communication requirements among the containers in the Virtual Cluster. Consider the bandwidth requirement between the virtual switch and container i as being B_i. Conservatively, B_i should be larger than the sum of the intra-group container traffic and inter-group traffic for container i. To successfully place a container group in the topology, we have to meet the container group's CPU and memory demands on the servers as well as the bandwidth requirements on every underlay links of the Virtual Cluster. Validating the server side resource requirement is

relatively straightforward. Therefore, we focus on the network link bandwidth reservation.

If we ignore the inter-group communications in Fig. 3.8 (a), the problem becomes placing n independent container groups (i.e., Virtual Clusters) in the data center without violating resource constraints. Considering a subtree structure T_i in the DCN topology as shown in Fig. 3.8 (b), the bandwidth of outbound link(s) between T_i and the remaining data center is the bisection bandwidth (for multiple paths) between T_i and the remaining data center. Under the general assumption, some containers have already been placed on subtree T_i. The pending container group to be placed on subtree T_i is group j (i.e., Virtual Cluster VC_j). Because of resource constraints at subtree T_i, the maximum number of containers that can be placed at subtree T_i is component a as shown in Fig. 3.8 (b). The component b of VC_j has to be placed at the remaining part of the data center. The maximum size of component a for VC_j is bounded by the residual bandwidth at the outbound link(s) [13]. If we know the residual bandwidth at the outbound link(s) of subtree T_i, we can calculate the maximum size of component a for VC_j.

With the maximum component a for VC_j at subtree T_i, VC_j is placed at the smallest left-most subtree that can support all m/n containers of group G_j. In such a way, all the independent container groups are successfully placed. The subgraph of the DCN with n active servers might not be able to support the demands of n Virtual Clusters, because the average server capacity is fully used, or because we assumed full network bisection when determining the number of groups n. Goldilocks increases the value of n by the size of one pod when there are one or more Virtual Clusters that have to be placed. Because containers with same Group id are placed in the smallest subtree, locality is assured.

3.4.2 Assignment with Inter-Group Communication

Subsequently, we place the n container groups as a whole. The ignored inter-group bandwidth B_{G1} to B_{Gn} in Fig. 3.8 (a) should be considered now. Suppose we try to place a 2-layer Virtual Cluster with only 2 container groups (G_1 and G_2), similar to Fig. 3.8 (a). Using the same example in Fig. 3.8 (b), the Group 1 has already been placed in the data center with component a (G_{1a}) in the subtree T_i and component b in the remaining data center. The next step is to place container group G_2 with component G_{2a} in subtree T_i. The containers of G_2 that can not be placed at T_i because of resource constraints are called component G_{2b}. The problem we try to solve is deciding the maximum size of G_{2a} based on the residual bandwidth at outbound link(s) [13] of subtree T_i.

In the independent group placing, the size of G_{2a} is only limited by the intra communication between component G_{2a} and G_{2b}. But now we need to consider the inter-group communication between G_{2a} and G_{1b}. Because G_{1b} of group 1 is placed outside of subtree T_i. The bidirectional bandwidth to be reserved on the outbound link(s) of subtree T_i, to satisfy group G_2, is given by R_{G_2}:

$$R_{G_2} = min\ (\sum_{\forall q \in G_{2a}} B_q, (\sum_{\forall r \in G_{2b}} B_r + \sum_{\forall s \in G_{1b}} B_s)) \tag{3.4}$$

$\sum_{\forall q \in G_{2a}} B_q$ is the sum of bandwidth for all the containers located in component G_{2a}. The required bandwidth at outbound link(s) to support G_2 could never be larger than the total bandwidth of component G_{2a}. $\sum_{\forall r \in G_{2b}} B_r$ is the intra-group communication between G_{2a} and G_{2b}. The inter-group communication between G_{2a} and G_{1b} is represented by $\sum_{\forall s \in G_{1b}} B_s$. If the sum of communication to the outside of subtree T_i (i.e., $\sum_{\forall r \in G_{2b}} B_r + \sum_{\forall s \in G_{1b}} B_s$) is smaller than the total bandwidth of component G_{2a}, we should reserve

$\sum_{\forall r \in G_{2b}} B_r + \sum_{\forall s \in G_{1b}} B_s$ bandwidth at the outbound link(s). Otherwise, we should reserve $\sum_{\forall q \in G_{2a}} B_q$ bandwidth.

Next, we consider the case with n container groups (G_1 to G_n). If our algorithm already placed G_1 to G_{k-1}, the next one to be placed is the container group G_k. The total bandwidth of component G_{ka} and the intra-group communication for G_k are similar to the terms at equation (4). Now, we focus on the inter-group communication to the outside of subtree T_i, given by the following equation:

$$\sum_{1 \le y \le k-1} \sum_{\forall r \in G_{yb}} B_r + \sum_{k+1 \le z \le n} \sum_{\forall s \in G_z} B_s \tag{3.5}$$

$\sum_{1 \le y \le k-1} \sum_{\forall r \in G_{yb}} B_r$ is the communication between component G_{ka} and all the component b of placed groups (G_1 to G_{k-1}). For the pending container groups G_{k+1} to G_n, to be conservative, component b has all the containers in that group and the size of component a is 0. So, $\sum_{k+1 \le z \le n} \sum_{\forall s \in G_z} B_s$ is the communication to all the unplaced container groups. By knowing the required bandwidth at the outbound link(s), the 2-layer virtual cluster can be assigned to the data center. Note that the bandwidth constraints on any links l inside T_i has already been satisfied when placing container groups on the subtree T_l inside tree T_i, as the container groups are first placed on the smallest tree in the topology [13].

3.5 Implementation

Our testbed consists of 16 compute nodes and a management node. All the compute nodes have a 32 core AMD Opteron 6272 CPU, 64G memory, 1G Ethernet NIC, 50G SSD and 12TB shared RAID file system. All the servers run CentOS 7.2. In order to sup-

port the Docker container migration, we customized the Linux kernel version 4.14.24 and CONFIG_CHECKPOINT_RESTORE is enabled in the kernel. In the network, a leaf-spine topology is achieved by using 3 HPE 3800 series switches with 48 1G ports each. One physical switch is divided into 8 virtual switches (distinct VLANs) to simulate the leaf switches. Every pair of physical servers is connected to a leaf switch. Leaf switches are connected by 2 spine switches in a full mesh.

All the applications are hosted in Docker containers (version 17.09.1-CE). Docker container migration is an essential function for the placement of containers from one execution to the next. At the end of each epoch, the containers are migrated to the new servers based on the algorithms in section 3.3 and 3.4. Goldilocks leverages the process checkpoint and restore technique [10, 66] to migrate containers between servers. The footprint of a process is typically stored as a disk image and then restored in the destination server. This process checkpoint & restore is a challenging task because a number of different pieces of information: namespace, iptables, cgroup and memory pages have to be obtained from the application's process tree. In our implementation, we use the Checkpoint Restore In Userspace (CRIU) [28] to achieve the Docker container checkpoint & restore.

The current CRIU has several limitations for container checkpointing. CRIU only checkpoints the file descriptor and inode information. We need to copy disk files and the Docker volume separately, for which we use rsync. Also, CRIU freezes the network connection by using iptable rules. It is essential that the same application-specific IP address exists at the destination server. In order to achieve seamless container migration, we adopt a similar policy as in VL2 [43]. The location-specific IP address is in subnet

192.168.0.0/16, but the application-specific IP address is in range 10.0.0.0/16. Node 16 in the cluster functions as a Docker swarm manager to also maintain the mapping between location-specific IP address and application-specific IP address. All the Docker containers are attached to the same overlay network, with tunnelling achieved by using VxLAN.

Apart from the 16 compute nodes, we have a distinct management node to implement Goldilocks. The management node has an out-of-band connection to the switch and in-band connection to all the compute nodes. Prior to mapping containers to servers, the real-time server and network utilization have to be measured. The server utilization is obtained by polling the Docker metric pseudo-files. Because packet encapsulation is used in the VxLAN overlay network, the inter-container communication pattern is obtained by using the IPTraf tool on servers to monitor the virtual Ethernet port for each container. We developed a migration controller in python to enforce migration rules, defining messages to orchestrate container checkpoint & restore. Servers can be remotely turned ON/OFF using an additional IPMI port. We estimate network power savings by determining which of the switches are idle.

3.6 Evaluation

We evaluate Goldilocks on a testbed implementation and compare it with a number of alternatives published in the literature, viz., E-PVM [8], mPP [112], Borg [111] and RC-Informed [27]. We also use large scale trace-driven simulation to further evaluate Goldilocks. For E-PVM, containers are placed on the least utilized machines. In mPP, containers are placed on servers in a First Fit Decreasing manner, where items are considered in decreasing

order of resource demand size. If the resource constraints are satisfied, the container is allocated to the server with least power increase per utilization unit. The difference between mPP and Goldilocks is that Goldilocks stops packing containers to the server if we reach the *Peak Energy Efficiency* (70% in the experiments), but mPP stops at the maximum server utilization (95%).

Borg is a cluster scheduling system from Google, including the capabilities of priority-based scheduling, task preemption, and task packing etc. [111]. In this chapter, we only implement the task packing algorithm of Borg, meant to reduce stranded resources. The last algorithm we compared is the bucket-based RC-Informed policy. In RC-Informed, the CPU resource is oversubscribed because the allocated resource for containers are usually not full utilized. Currently, the CPU resource is 125% oversubscribed in RC-Informed [27]. To be fair, only the placement algorithms in the alternatives are implemented on our Docker container-based testbed, independent of whether it is a virtualization-based system (e.g., mPP and RC-Informed) or a Linux container-based system (e.g., Borg and E-PVM).

3.6.1 Testbed Results

Twitter Content Caching on Wikipedia Trace Pattern. We start with an experiment using the Twitter content caching workload. We set up a fixed, total number of 176 containers in the testbed, based on the goal of ensuring that the average server utilization for E-PVM is 32%, in line with prior observations [74, 60, 69]. In this experiment, the front-end container queries for a set of twitter terms from the Memcached container. If there is a hit for the query in the memory, a success occurs for the get operation.

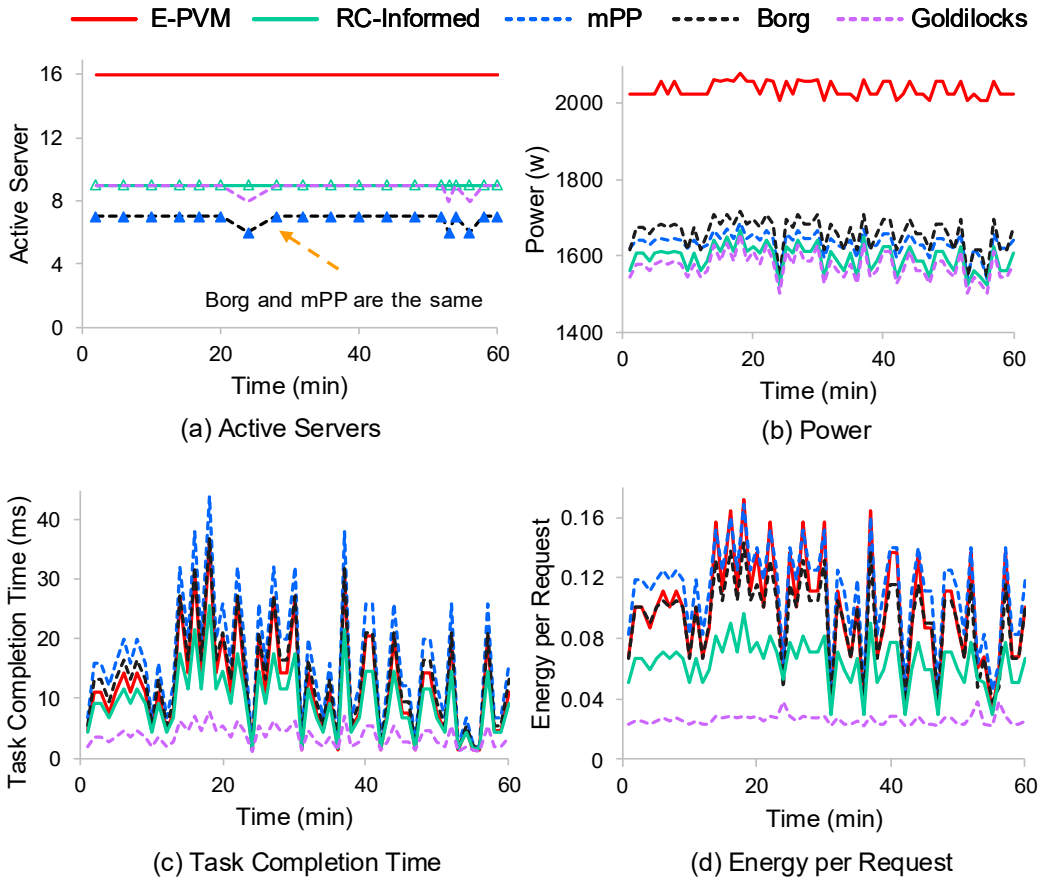


Figure 3.9: Twitter content caching on the Wikipedia trace pattern.

Fig. 3.9 shows the time-varying results across the alternatives and Fig. 3.10 shows the average values. The query pattern for the request per second (RPS) rate follows the Wikipedia trace [107]. The RPS ranges from 44K to 440K across the entire testbed. In Fig. 3.9 (a), all the servers are active in E-PVM. Most of the time, Goldilocks and the bucket-based RC-Informed need 9 active servers, compared with 7 active servers in Borg and mPP, because servers in Goldilocks are limited to a maximum of 70% utilization. All the placement policies, except E-PVM, might need less active servers at low RPS, as the resource demands per container decrease. Goldilocks in Fig. 3.9 (b) consumes the least amount of

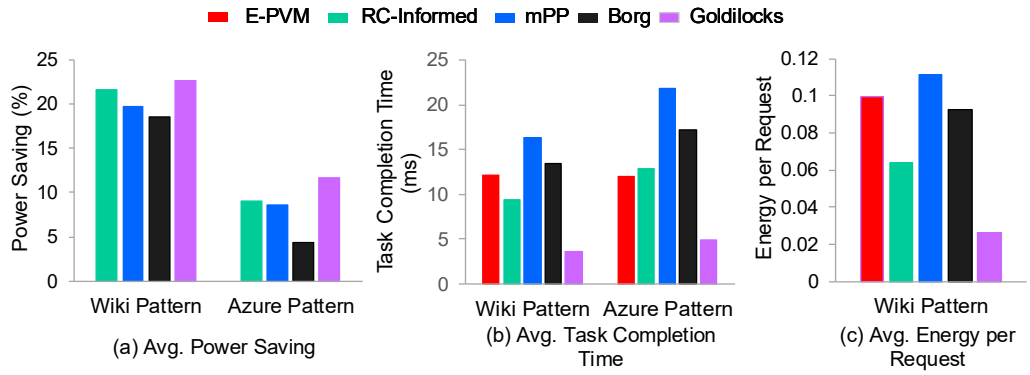


Figure 3.10: Average power saving, task completion time and energy per request results for 2 trace patterns.

power over the 60-mins experiment, because it seeks to achieve the *Peak Energy Efficiency*. The average power savings of mPP, Borg, RC-Informed and Goldilocks, compared with E-PVM on the Wiki pattern, are given in Fig. 3.10 (a). We can observe that Goldilocks saves 22.7% of the entire data center's power and outperforms all the alternatives.

More importantly, task completion time of queries improves significantly with Goldilocks in Fig. 3.9 (c). Goldilocks has the smallest task completion time during the 60-mins experiment. The average task completion time results are plotted in Fig. 3.10 (b), with Goldilocks giving the lowest result of 3.67ms. Among the alternatives, RC-Informed has the smallest task completion time of 9.44 ms. It is a bucket-based scheduling policy and has the lowest server utilization, compared with Borg and mPP. The average task completion time of Goldilocks is only 33% of RC-Informed, as we place the query generator and responder closely in the testbed.

The next important consideration is the energy consumed per request. Because of *Peak Energy Efficiency* (i.e., less power consumption) and locality-focused graph partitioning (i.e., shorter task completion time) in Goldilocks, it has the lowest energy per request

results in Fig. 3.9 (d). We first take a look at the alternatives. On average, RC-Informed has the best energy per request result of 0.06, as shown in Fig. 3.10 (c). The energy per request result of Goldilocks is 3 times of better than RC-Informed, because of least power consumption and shortest task completion time in Fig 3.9 (b) and (c).

Rich Mixture of Applications on Azure Trace Pattern. In the previous experiments, we had a single Twitter workload and the number of containers was fixed at 176 and we vary the RPS from 44K to 440K. We now consider a rich mixture of applications, as is typical in a cloud data center and reflected in workload trace from the Microsoft Azure cloud [27]. We compare Goldilocks with the alternatives under this workload mix in terms of total number of containers needed in the testbed. The RPS for the containers supporting Twitter content caching is set at 2K for each connection. The total RPS for the entire set of services depends on the number of containers in the data center. In addition to the Twitter caching workload, we add 6 other background applications: the Apache Solr Search Engine, a movie recommendation system on Spark, Hadoop, a page rank on Spark and Cassandra database. The total number of containers ranges between 149 and 221, following the pattern found in Microsoft Azure trace [27]. We achieved this by stopping existing containers and launching new containers in the testbed. For the baseline E-PVM, the average server utilization gets to be as high as 54%.

Fig. 3.11 (a) plots the number of active servers for E-PVM, mPP, Borg, RC-Informed and Goldilocks. Because of the mix of applications, packing of containers does result in higher fragmentation. Goldilocks needs 2 more active servers to serve the same number of containers compared to Borg and mPP, because of 70% server utilization in

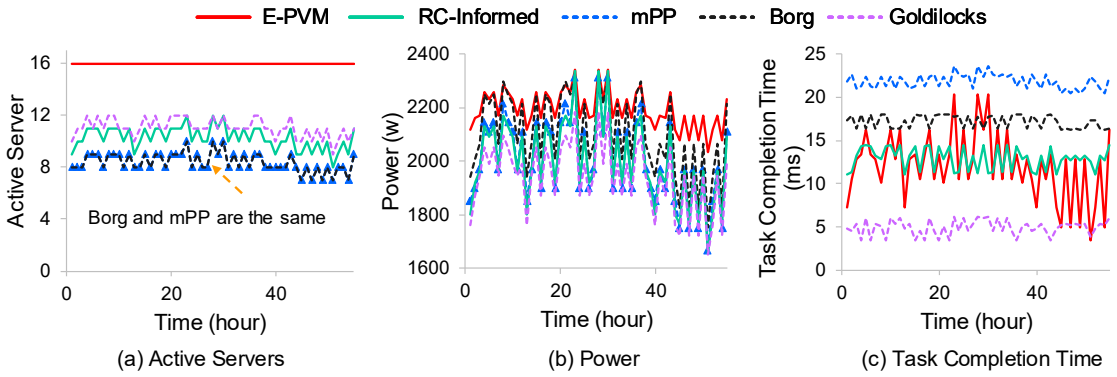


Figure 3.11: Rich mixture of applications on the Azure trace pattern.

Goldilocks. In Fig. 3.11 (b), Goldilocks again consumes the lowest total data center power. Compared with the baseline E-PVM, the power saving of Goldilocks increases from 6.6% to 18.8% when the total number of containers in the testbed decreases to 149. We observe that mPP, Borg and RC-Informed consume similar amount of power as the baseline E-PVM, at about 55% average server utilization. The reason is that these scheduling policies keep packing containers until reaching maximum server utilization, where the server power increases non-linearly. At the same average server utilization, Goldilocks consumes 6.7% less power compared with the base line E-PVM. Similarly, we plot the average power saving results compared with E-PVM in Fig. 3.10 (a). On average, Goldilocks achieves 11.7% power saving and the best one in alternatives (RC-Informed) has power saving of 8.9%. Fig. 3.11 (c) compares the task completion times. Goldilocks has the shortest task completion time, by far, for Twitter queries. As shown in Fig. 3.10 (b), the average task completion time for Goldilocks is 4.9 ms. The best, among the others compared is E-PVM, but even that has 2.5 times higher average task completion time.

3.6.2 Simulation Results

In addition to the testbed implementation, we also performed a flow-level, large scale simulation with a 28-ary fat tree topology, with a total of 5488 servers and 980 switches. The power model we used for servers is Dell Power Edge R940 [103] and the power model for switch is HPE Altoline 6940 [86]. The flow level simulation uses the processing time distribution for search queries based on a micro-benchmark of measurements made in our testbed. There are a total of 49392 containers in the topology, targeting a 20-30% server utilization [74, 60, 69] for the baseline E-PVM. We then simulated Goldilocks, mPP, Borg and RC-Informed to get the power savings and improvements in task completion time.

Since the trace [6] used for the simulation has only the flow level information, we obtain the resource demands on the servers through experiments in our testbed with a workload matching the trace's network traffic pattern and then measured the server resource utilization. In the trace, there are two kinds of traffic: search queries with flow sizes ranging from 1.6KB to 2KB and background update traffic with 1MB to 50MB flow sizes. We assume for the purposes of our simulation that the background traffic is Hadoop traffic because the URL crawling in search is based on the Map-Reduce framework.

In Fig. 3.12 (a), we deploy the Apache Solr search engine in the testbed and vary the search request rate. Since the maximum number of connections per ISN in the trace is 120, we increase the request rate up to 120 RPS. The memory usage stays at 12G throughout. Fig. 3.12 (b) shows the background update traffic generated in our testbed deployed as a 16-node Hadoop cluster running the Facebook job trace [20]. The aggregate traffic and corresponding CPU utilization are then measured. Each color in the figure represents the

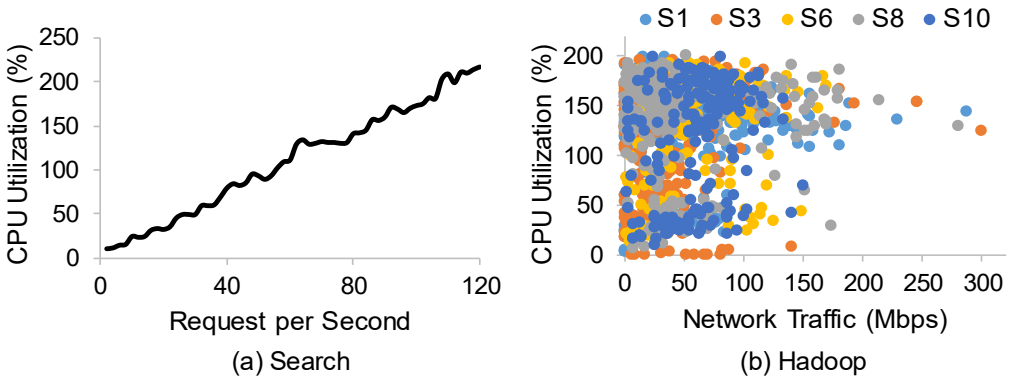


Figure 3.12: Part (a), measured CPU utilization for Apache Solr search engine. Y axis is defined as the sum of all CPU core's utilization. Part (b), CPU utilization for varying traffic rates, with 5 servers in a 16-node Hadoop cluster running Facebook trace.

data from one of the slave nodes in the Hadoop cluster. As shown in Fig. 3.12 (b), there are multiple dots with the same network traffic rate (X-axis value). In the simulation, a randomly chosen CPU utilization (Y-axis value) is chosen for a selected network traffic rate. The resource demands on the servers is the result of the sum of these 2 applications.

Fig. 3.13 (a) shows the number of active servers over a period of 88 hours. Because E-PVM always chooses the least utilized server, all of the 5488 servers in the topology are active. Borg and mPP pack the containers to achieve 95% server utilization and both of them have the least number of active servers. RC-Informed is a bucket-based scheduling policy, and the number of active servers is constrained by the total reserved resources for each container rather than the real-time resource utilization in the simulation. That is why RC-Informed needs 2358 active servers most of the time. Although Goldilocks needs more active servers compared with the other container packing policies, it consumes the least amount of power, as shown in Fig. 3.13 (b). Goldilocks stops packing containers when the server reaches its *Peak Energy Efficiency* (70% in this simulation). The baseline, E-PVM,

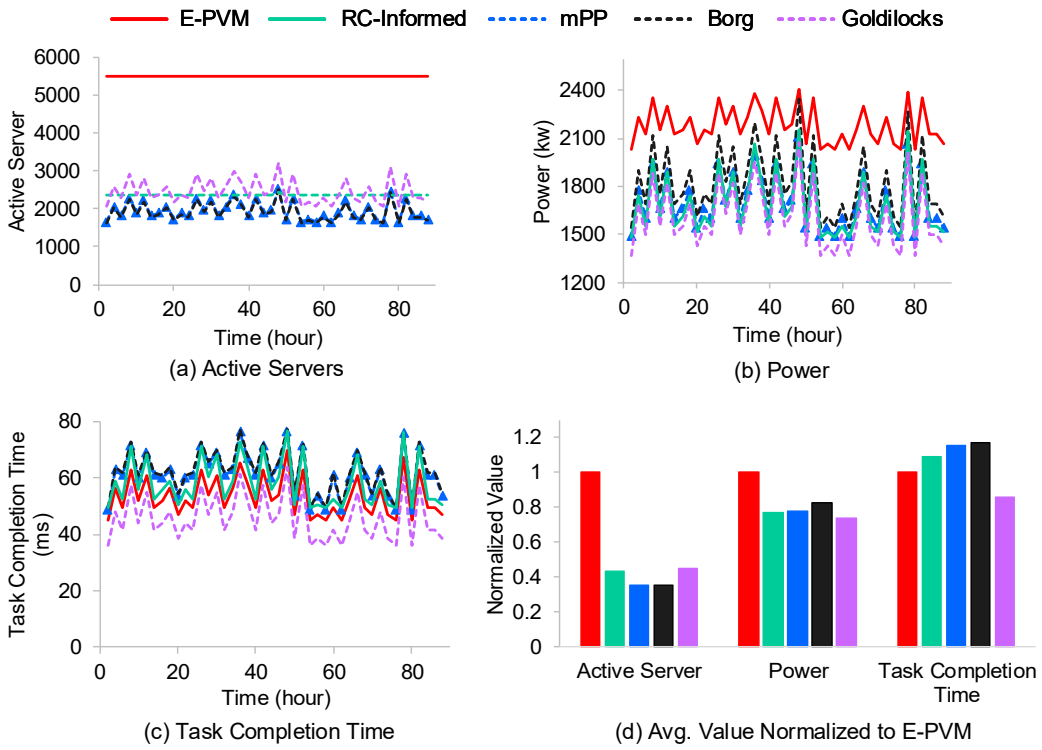


Figure 3.13: Trace-driven simulation results.

consumes the highest data center power because of no saving in static server power. The power consumption of mPP, Borg and RC-Informed are similar, within 150kw of each other.

Finally, we take a look at the task completion time of search queries in Fig. 3.13 (c). Goldilocks has the shortest task completion time because of large server headroom and good locality. The average server load on E-PVM is around 26% to 40%. mPP, Borg and RC-Informed target at 95% [27] server utilization. This high server load contributes to their increased task completion time compared with E-PVM. In Fig. 3.13 (d), we see the average values of active server, power consumption and task completion time. All the values are normalized to the values of baseline E-PVM. Goldilocks has the lowest power consumption (27% power saving compared with E-PVM) and shortest task completion

time (0.85 of E-PVM). Compared with the best alternative for each performance metric, Goldilocks consumes 5.2% less power than RC-Informed and achieves 15% shorter task completion time than E-PVM.

Chapter 4

DREAM: Distributed Traffic Consolidation in the DCN

4.1 Introduction

Traffic consolidation shifts network flows to a minimal number of active switches and turns off idle switches and links to save power. State-of-the-art traffic consolidation frameworks [49, 113, 125] are not responsive to traffic variation because of the long computation times involved with the optimization in a centralized manner and also depends on the limited refresh rate for the flow-level statistics from network switches. Poor responsiveness results in link congestion or packet drops in the network as well as poor adaptation in terms of managing energy consumption. This motivates us to design a distributed energy-aware traffic management framework for the DCN. The distributed design can quickly adapt to traffic fluctuation in the DCN, and promises to fully take advantage of every energy saving

opportunity when the network load ebbs. What is more, it also can move out traffic from congested paths in a more timely manner. In such cases, it reduces the packet drop rate and latency.

Our distributed design, DREAM, uses the existing primitives in hardware switches such as Explicit Congestion Notification (ECN) marking [41, 6, 92] to report link congestion to DREAM's distributed agents in servers. The ECN marking in DREAM is also beneficial for controlling latency for an application's flow. The distributed agents monitoring ECN feedback reroute traffic when a path is congested, but at a flowlet granularity. DREAM reacts before significant queuing or packet loss occurs.

The basic scheduling unit in DREAM is flowlet (i.e., a burst of packets) [5]. This enables it to take advantage of the multiple active paths in the DCN without any modification to the host protocol stack. Flowlet scheduling also reduces fragmenting of the link capacity and allows for better energy savings. For each active path, a value representing the probability of sending next flowlet on that path is maintained. The selection probability value is increased additively after successfully sending a flowlet over an uncongested path, and decreases substantially when observing ECN feedback indicating congestion on the path. Increasing the selection probability of left-most path correspondingly reduces the selection probability for the right-most path, thus eventually resulting in traffic consolidation. Prior work [21] on proving that the equilibrium point for Additive Increase Multiplicative Decrease (AIMD) can be re-used to show that its use in our design can result in a stable operation.

Finally, we propose a packet encapsulation format in the Open vSwitch to achieve

explicit path control. In prior works [49, 113, 125], the routing path is decided by the centralized controller. DREAM has to enforce explicit routing path in a distributed manner. Inspired by Xpath [52], we pre-install the forwarding rules based on a compressed path ID at switches. At the traffic source, the distributed agent encapsulates packets and inserts the path ID in an encapsulation header. Switches match on the path ID along with other header fields to route the flows appropriately. To minimize encapsulation overheads, we re-use primitives already in Open vSwitch.

DREAM is implemented in a testbed DCN with a leaf-spine topology. The testbed evaluation uses the Wikipedia traffic trace [107] and more bursty Facebook MapReduce traffic trace [19]. We show that DREAM on average achieves at least 15.8% DCN energy saving, while CARPO and ElasticTree produces 11.6% and 8.4% energy saving, respectively. The packet drop ratio in DREAM is less than 0.01% while the best among the alternatives, ElasticTree [49], has 0.19% drop ratio on Facebook trace and 0.85% drop ratio on Wikipedia trace. Finally, DREAM has at least 30% shorter application-level latency compared to the alternatives. Our major contributions include:

- To improve responsiveness and scalability, we consolidate traffic to a subset of the DCN in a distributed manner, to achieve energy saving. The distributed agent is implemented in Open vSwitch without any modification to the host protocol stack or the data center network switches.

- We propose to reroute traffic based on ECN feedback, with congestion information obtained every RTT. The more frequent, accurate feedback of incipient congestion on the precise path a flowlet traverses helps achieve much better network latency.

- Network traffic is scheduled at the granularity of flowlet to take advantage of multiple active paths in the DCN, without any host side modification. We propose a multi-path flowlet scheduling algorithm for the energy efficient DCN.
- We propose a packet encapsulation format for explicit path control in the network.
- We demonstrate the effectiveness of DREAM in a testbed implementation with production switches.

4.2 Background and Motivation

4.2.1 Energy Efficient Data Center Networks

Data centers host tens of thousands of servers and consume tens of Megawatts of power [108]. Power management on servers is an important component, and the use of techniques such as DVFS [69, 108, 130, 60] and VM migration [27, 112, 111] already seek to save power. ElasticTree [49] and a number of other following works [113, 126, 125] seek to make the data center *network* energy proportional as well, to complement the power management on servers.

ElasticTree [49] converts the energy management of DCN into an augmented Multi Commodity Flow (MCF) problem and then solves it by using Linear Programming (LP). Typically, the LP program runs at the centralized Software Defined Network (SDN) [18] controller. The SDN controller leverages the Openflow [73] protocol to fetch traffic statistics from the hardware switches periodically. The future bit rate of flows is predicted based on the traffic history in the last epoch, while providing for a safety margin [49]. Based on

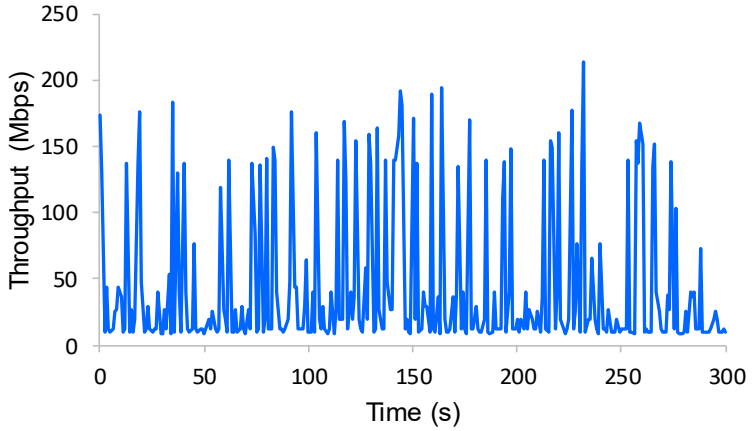


Figure 4.1: Traffic in the DCN has high variation, as shown for an example from the Microsoft search trace.

these traffic statistics, the centralized optimizer in the SDN controller runs every epoch to determine which subset of the DCN should be active. The epoch length needs to be longer than optimizer's computation time and more importantly longer than the rate at which the traffic statistics are reported by the hardware switches. The centralized optimizer outputs the updated routing path for each flow to achieve optimal network energy savings, which is enforced by the centralized SDN controller.

4.2.2 Traffic Variability and Responsiveness

Network traffic in data centers can be highly variable [14]. Some applications, such as MapReduce [34], generate bursty traffic in a short time period, especially for the data shuffle phase. In Fig. 4.1, we plot the traffic variation of Microsoft Search trace [6] spanning a 5 min. interval. For example, the throughput at 34th second is 8.8Mbps but bursts up to 183Mbps at the next second.

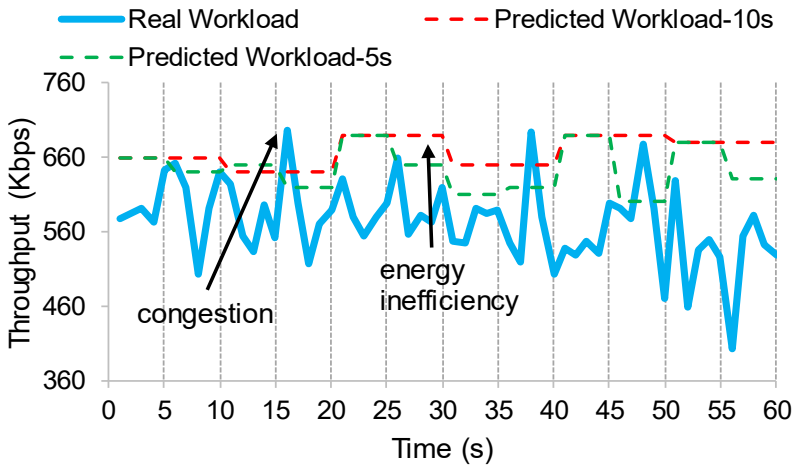


Figure 4.2: Poor responsiveness to traffic variation incurs link congestion or energy inefficiency.

In the ideal case for an energy proportional DCN, the energy consumption should adapt to the traffic variation. Otherwise, we suffer either link congestion or energy inefficiency. As we discussed in the previous subsection, the responsiveness of the energy saving framework depends on the epoch length for traffic scheduling. In Fig. 4.2, we give an example to show how the epoch length might impact energy saving as well as link congestion. The blue line represents one network flow in the data center. Its throughput varies over the 60 seconds shown. First, we set the epoch length for traffic consolidation at 10s. The red line is the predicted throughput based on the 90th-percentile [113] throughput in last epoch. Then, the centralized controller will reserve network bandwidth based on this predicted traffic data rate. Due to prediction errors, the observed flow's throughput might be larger than the predicted value, as shown in the interval 15s - 20s. But the centralized controller will reroute traffic only at the end of current epoch (i.e., at 20s). Packet drop and TCP retransmission could occur. A shorter epoch length promises to achieve better energy

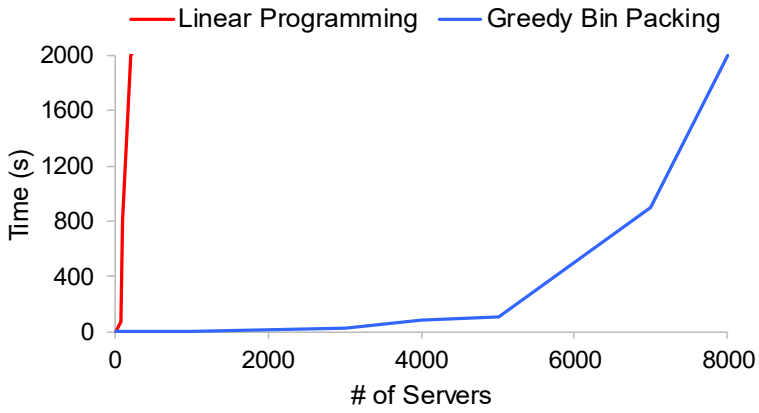


Figure 4.3: Computation time of linear programming and greedy bin packing for various network topology sizes.

savings. We also plot the predicted workload for 5s epoch length (green line). During the interval 15s - 20s, when the rate of the flow decreases, the optimizer with 5s (smaller) epoch length will capture this energy saving opportunity, which will be missed if a longer epoch is used.

The epoch length is limited by two considerations: the computation time of the optimizer and the measurement frequency to obtain the traffic statistics. Modeling the traffic consolidation as a linear programming model has high computation complexity [109]. Although a heuristic algorithm has been proposed to accelerate this [125, 49], the computation time can still be high for large data center network topologies. Fig. 4.3 shows the computation time of linear programming and heuristic greedy bin packing [49] for different data center sizes. Greedy Bin Packing still takes more than 1000s for a medium size data center. We realize that this can be speeded up with a faster server. However, the epoch length is still constrained by measurement frequency of traffic statistics. In the centralized design, the controller has to poll the hardware switches for traffic stats using protocols such

as SNMP [100], sFlow [95] and Openflow. The measurement frequency cannot be very high due to measurement overheads. For example, a hardware switch in our testbed only updates the traffic metrics for Openflow [73] protocol every 20 seconds [51]. That means the epoch length has to be 20 seconds or even more.

4.3 DREAM Design

DREAM is a distributed traffic consolidation framework operating at a flowlet-level to achieve energy savings in data center networks. Its distributed and adaptive design makes it responsive to latency-sensitive applications and rapidly adjusts to bursty network traffic. The flowlet-level scheduling by agents at endpoints takes advantage of the multiple paths potentially available in data center networks without any modifications to existing data center infrastructures. Flowlet-level scheduling with the much shorter decision-making epochs enables DREAM to fully utilize every energy saving opportunity, as the workload varies. Finally, DREAM has lower packet drop ratio and application-level latency by adopting ECN for flowlet path selection.

4.3.1 Overview

The design of DREAM is shown in Fig. 4.4. Instead of using a centralized controller to do data center wide traffic engineering [49, 125, 113], we use distributed agents at each end-system to work cooperatively to achieve energy saving in the DCN. The distributed agent at the servers is a shim layer between the host's protocol stack and the data center networking fabric. We use the Open vSwitch [89] virtual switch as an example of

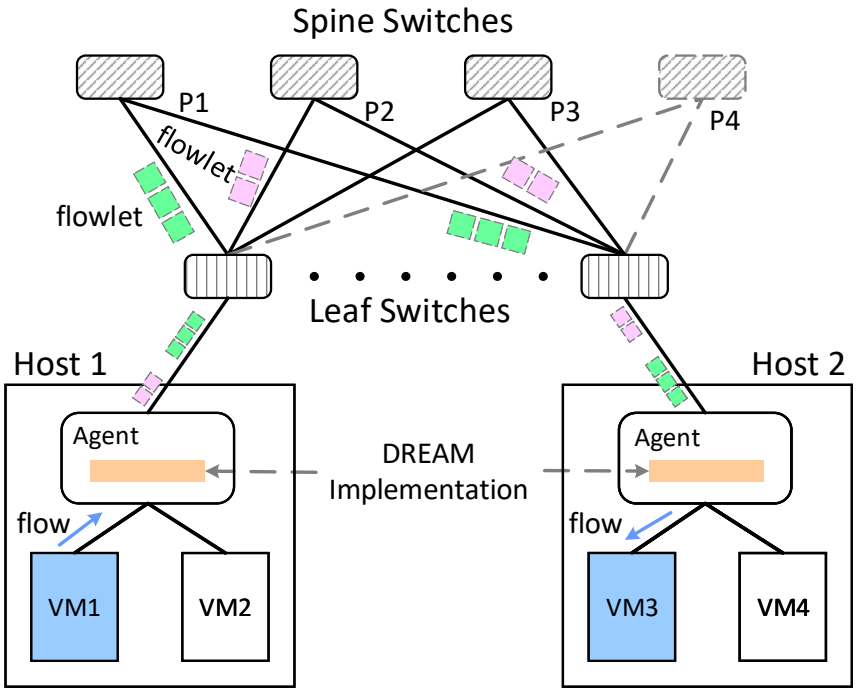


Figure 4.4: The design overview of DREAM.

this shim layer, since it is widely used in virtualization environments for software-based switching and forwarding and network automation. The distributed design in DREAM has better scalability and is more responsive to bursty traffic. In addition, there is no need to modify host applications or the switches in the DCN.

For example, consider Fig. 4.4 where there are 4 paths ($P1$ to $P4$) between Host 1 and Host 2. To save energy in the DCN, we seek to consolidate traffic to a minimal number of paths, without violating performance (i.e., latency) requirements, and then turn off idle switches and links (e.g., the switch and links along path $P4$). At a high level, DREAM consolidates traffic to a part (e.g., 'left-most' for a Leaf-Spine topology) of the DCN topology. By 'left-most', we mean the active switches of a single layer at the structured

DCN topology are chosen from left to right [49]. To avoid potentially large wake up times for OFF switches impacting flows, a few backup paths are reserved in the DCN topology, thus slightly over-provisioning DCN capacity. Path $P3$ in Fig. 4.4 works as a backup path.

Reducing network delay is very important for latency-sensitive applications. Similarly, avoiding packet drops is important to maintain TCP throughput [6, 5]. Hence, we leverage the commonly used ECN feedback for congestion response. We also use ECN to steer traffic to an alternate path when one is congested. Idle path such as $P4$ can be turned off to save DCN energy. But, there are still 3 active paths between Host 1 and Host 2 in Fig 4.4. When the end-systems are not the bottleneck, the ability to use multiple paths between a sender-receiver pair can improve throughput [114]. DREAM splits a flow across multiple active paths in the DCN at the flowlet granularity. For example, we split the flow between VM1 and VM3 among active paths $P1$ and $P2$. A flowlet is a burst of packets of a flow [5], where the inter-flowlet time gap is much longer than the inter-arrival times of packets within a flowlet. The traffic sent on each DCN path is a group of flowlets rather than a larger flow. Thus, DREAM reduces the fragmenting of bandwidth capacity on each active path, thus improving multiplexing efficiency on each path (i.e., 'packing' more traffic).

At each distributed agent, we maintain a probability for selecting each active path. The path selection decision for an incoming flowlet is made based on the selection probability for each path. We dynamically change the path selection probability for each active path based on network conditions. For example, in a Leaf-Spine network, we indirectly consolidate traffic to the left-most paths by increasing the selection probability for the left-most path and reducing the selection probability for the right-most path. When ECN marks are

received for traffic on a particular path, the selection probability for that path is reduced to mitigate congestion. Only a portion of flowlets are moved to different less-utilized path (that has a higher selection probability), rather than moving the entire flow, as used in prior distributed designs [109, 128]. This significantly reduces the oscillations [57] typically observed in distributed designs for DCN energy reduction.

Unlike previous centralized designs that do not consider the new arrivals of flows [49, 113, 125], DREAM explicitly deals with new arrivals of flows. DREAM first places a new flow on the right-most path (i.e., $P2$ in Fig. 4.4) and then gradually moves traffic to the left part of active DCN topology. This avoids congesting paths with existing flows and only migrates the new flows to the favored left-most paths as network conditions warrant. Previous works factor only existing flows as a variable in a linear programming formulation and do not deal with online flow arrivals.

Finally, we use packet encapsulation by the distributed agents to achieve explicit path control. In DREAM, the forwarding rules at hardware switches are set up by re-utilizing the schemes described in Xpath [52], with highly compressed rules to save Ternary Content-Addressable Memory (TCAM) resources in switches. The path IDs are inserted into the outer packet header by the distributed agent. Hardware switches forward packets accordingly, based on the path ID. For the sake of efficiency, we utilize the existing primitives in Open vSwitch for the packet encapsulation.

4.3.2 Distributed Agents

In DREAM, the distributed agent is implemented at the shim layer between host's protocol stack and the DCN's networking fabric, for the sake of easy deployment. Open

vSwitch (OVS) has already implemented the Generic Routing Encapsulation (GRE) and VxLAN packet encapsulation for tunneling, which we utilize in DREAM. As all the traffic goes through the OVS data plane, it is an ideal place for the distributed agent of DREAM.

The basic scheduling unit in DREAM is a flowlet. Packet reordering is minimized if the idle interval between flowlets is more than the maximum difference between the delays across different paths [5]. Moreover, the TCP protocol stack has increasingly supported packet reordering to support multi-path, and we take advantage of it, when it does occur across flowlets.

In DREAM, we consolidate flowlets on a path until we observe ECN on that path. While ECN is usually used for network congestion feedback [41, 6, 92], we leverage it to steer packets to avoid congested paths in the DCN. ECN provides feedback of incipient congestion and thus reduces packet drops and latency by having ECN-enabled TCP connections respond to ECN marks. Hardware switches with ECN capability use an Active Queue Management (AQM) at the interface level to mark the CE bit, which is reflected back in the ECE bit of TCP ACK packets. Our distributed agent monitors the ECE bit in the TCP header to sense path congestion, and moves flowlets from congested paths to alternate under-utilized paths.

Fig. 4.5 shows the implementation of DREAM in the data plane of Open vSwitch. The probability of sending a flowlet on each possible path (columns) is maintained in a table for each flow identified by the 5-tuple (protocol type, src.IP, dst.IP, src.port and dst.port packet header fields). This selection probability is used to select the path ID which is stored as the Selected Path ID in an additional column. The selected path ID is for the currently

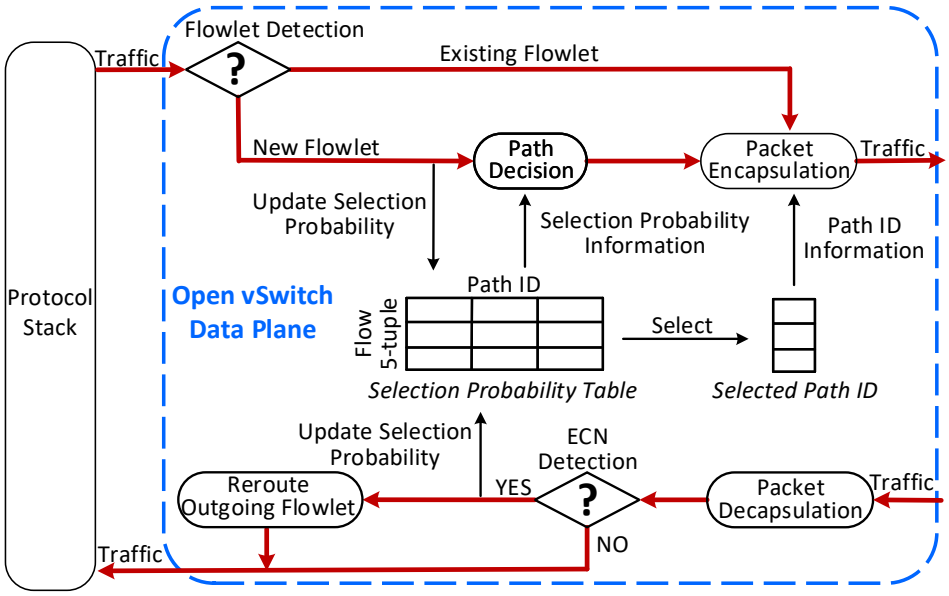


Figure 4.5: DREAM implementation in the data plane of Open vSwitch (OVS).

identified flowlet of the flow. Note that each cell in the Selection Probability Table is a probability value but each cell in the column of Selected Path ID is an integer ID.

First, we introduce the workflow of the outgoing traffic from the host's protocol stack to the DCN. Based on the packet inter-arrival time, the Flowlet Detection module will decide if we have found a new flowlet. The packets of an existing flowlet will be directly encapsulated with path ID before sending it out on the wire. Upon finding a new flowlet, the selection probability of previously completed flowlet has to be updated. The details on how to change the selection probability are described in section 4.3.3. We then choose a path for the new flowlet based on the path selection probability. As a last step, packets are encapsulated.

Second, the processing of incoming traffic from the DCN is shown at the bottom of Fig. 4.5. After decapsulating the packet, we check the ECN flags. If congestion has been

detected on a path (indicated by the ECE bit of ACK packets), DREAM will update the selection probability of that path and reroute the flowlet to an alternate under-utilized path before forwarding the packet to the host's protocol stack. Otherwise, the incoming traffic is directly forwarded to the host's protocol stack.

4.3.3 Scheduling Algorithm

Whenever we detect a flowlet in DREAM, the distributed agent makes a decision to choose a path for this flowlet, with all the packets in the flowlet following the same path. DREAM schedules the flowlet on a path based on the selection probability, f_i for path i. We seek to set the value of f_i such that we don't observe ECN on the active routing path. In Fig. 4.6, we give the state machine for flowlet scheduling.

Over-utilized. We start our algorithm description with the system being in equilibrium, where the traffic is consolidated onto a few active paths and none of the paths see congestion (as observed with ECN). Traffic fluctuation is common in DCNs [14, 93]. When we sense that one of the active paths is congested, some traffic have to be moved to another under-utilized path. Unlike prior works [49, 113, 125], that move the entire TCP flow every time epoch, DREAM does this at the flowlet-level. This enables DREAM to be more responsive and dramatically eliminate traffic oscillation. A subset of the flowlets on a congested path are re-routed, thus alleviating congestion but also ensuring that there is no traffic oscillation [57]. Traffic are routed on uncongested paths by adapting the probability of sending the subsequent flowlets on those paths.

In DREAM, the moving average of ECN history is maintained for each path. e_{new} is the fraction of marked packets with ECN bits in the current time window. The ECN

marking history e is smoothed using Exponentially Weighted Moving Average (EWMA) as $(1-\alpha) * e_{old} + \alpha * e_{new}$. α is the weight for new fraction of ECN markings. Thus, the distributed agents can react to link congestion at RTT timescales. Since the RTT in data center networks is usually a few microseconds [78], it enables DREAM to be very responsive. When the agent detects ECN on path i, the selection probability f_i is decreased by $X\%$. $X\%$ is defined as $f_i * e/2$.

Under-utilized. To save energy, we seek to consolidate traffic to one portion (e.g., left-most active paths) in the DCN. Let us consider a TCP flow with 4 groups of flowlets routing on 4 different active paths. The probability of sending the next flowlet on each of the paths is f_1, f_2, f_3 and f_4, respectively. For each path i (=1, 2, 3, 4), we increase the value of f_i if the distributed agent successfully sends a flowlet on path i without observing any congestion feedback signals. The probability increase follows the Additive Increase rule, i.e., increase f_i by a fixed amount (e.g., 1%). Since we guarantee that the sum of f_1, f_2, f_3 and f_4 is 100%, we correspondingly decrease the value of right-most active path (i.e., path 4) by 1%. In this design, flowlets are more likely to be routed on the left-most available active paths. One major concern is that the Additive Increase of f_i will finally result in queue build-up on path i. Then, we will revert to the case of over-utilized paths, in which we move a subset of flowlets to the under-utilized right most paths. The motivation of this design is to take advantage of every energy saving opportunity during the period when there is traffic variation and to react each traffic burst as well, just like TCP varies the congestion window.

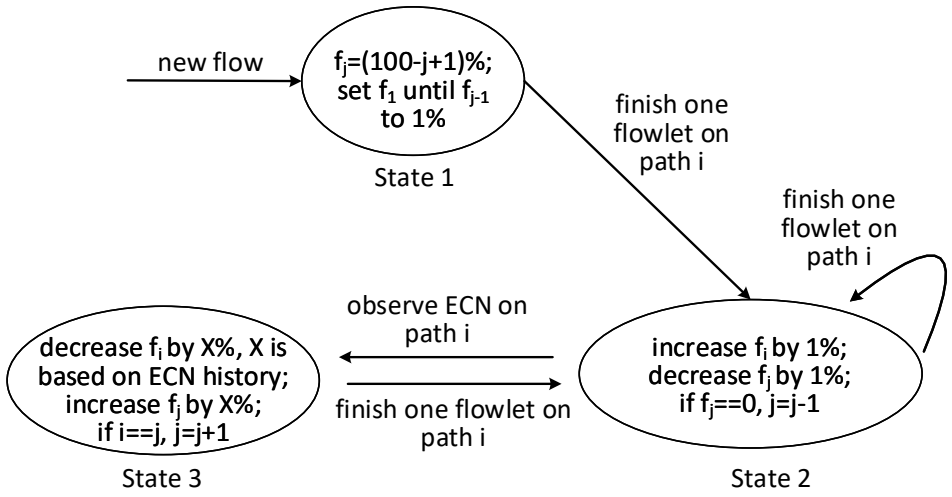


Figure 4.6: The state machine for flowlet scheduling.

New Arrivals. Unlike ElasticTree [49] and CARPO [113], which are offline solutions for a fixed set of flows, DREAM supports new flows for an online solution. Newly arriving flows are initially routed over the right-most active path j, to avoid congesting the heavily utilized left-most paths (1 to $j-1$) in the network. As the new flow's flowlets generate traffic, they gradually move to the left-most active paths (1 to $j-1$) at the probability of 1%, as long as these paths are not congested. If we successfully send a flowlet on path i, then we enter the under-utilized case for path i.

Finally, we show the state machine for our flowlet-level scheduling in Fig. 4.6. The state machine starts with the arrival of a new flow. First, we place all the flowlets on the right-most path j. The selection probability f_j for right-most active path j is (100-j+1)%. To gradually consolidate traffic to the left-most paths, we set f_1 until f_{j-1} to 1%. For any path i completing the transmission of one flowlet, we increase its probability value f_i by 1%. Correspondingly, the selection probability value of right-most path j decreases by 1%. When observing ECN feedback on path i, we enter state 3. Here, the distributed agent

decreases the probability value f_i by X% depending on the smoothed value of the ECN feedback. After that, each probability value f except the right-most one f_j recovers its probability value to reach a dynamic balance.

4.3.4 Packet Encapsulation

In data centers, the switches locally decide the next hop using Equal Cost Multiple Path (ECMP) or Valiant Load Balancing (VLB) [43]. In energy-proportional networks, we need to route traffic over a minimal number of switches and links thus turning off idle networking components. For this, explicit path control is essential.

There are many approaches such as Myrinet [83] and Multi Protocol Label Switching (MPLS) [82] to achieve the explicit path control. MPLS uses distributed protocols such as CR-LDP [29] or RSVP-TE [94] to populate labels in the network to set policy-based paths, but can be complex [52]. Previous centralized traffic consolidation frameworks [49, 113] leverage the SDN controller to insert forwarding rules at the Openflow switches, but the refresh rate on the Openflow TX/RX counters in hardware switches can be a limiting factor for responsiveness.

Xpath [52] proposes an efficient way to do explicit path control in the DCN. It uses a compressed hierarchical path ID to save TCAM space in the switch. A data center operator pre-installs the forwarding rules at hardware switches based on this path ID. In our design, we use packet encapsulation at DREAM's distributed agents to achieve explicit path control by making path selection decisions and putting the path ID in the encapsulation header. Fig. 4.7 gives the packet format of our design. Similar to the existing encapsulation

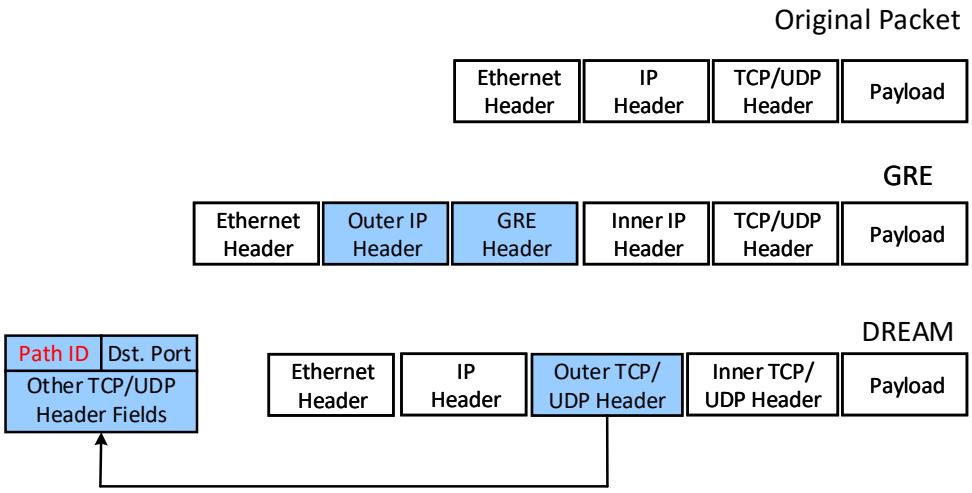


Figure 4.7: The packet encapsulation format in DREAM. Our encapsulation header is between the IP header and Inner TCP/UDP header. The original Src. port field of Outer TCP/UDP header is replaced by Path ID.

protocols such as GRE in Open vSwitch, DREAM's encapsulating packet header comes between the original IP header and the TCP/UDP header (shown as Inner TCP/UDP header). We directly copy the fields in inner TCP/UDP header to the outer TCP/UDP header, maintaining the rest of the layer 4 information, except for using the source port number in the outer header to store the path ID. Existing hardware switches match on this path ID, and require no modification. The original source port number is stored at inner TCP/UDP packet header which will be used by the end-system protocol stack after packet decapsulation at the destination server.

4.4 Implementation

DREAM is implemented on a testbed with 8 blade servers and 6 switches. The servers and switches are hosted at two different racks. In the network, we create a 2-layer leaf spine DCN topology. There are two types of switches in the network: HPE 3800

J9574A leaf switches and Arista 7050X-72Q spine switches. A pair of 2 servers as a group is connected to a leaf switch.

Each server has a 32 core AMD Opteron 6272 CPU, 64G memory, 1G Ethernet Network Interface Card (NIC), 50G SSD and 12 TB shared RAID file system. The TCP protocol on our CentOS 7.2 operating system is TCP Cubic. We also disable the Nagle algorithm on the operating system to have more accurate latency results. ECN is enabled in the Linux kernel. We take control of the host NIC by a modified version of Open vSwitch 2.4, in which we implement the distributed agents of DREAM. For Open vSwitch, most of codes of DREAM are implemented at the data plane module. We re-utilizes the sk_buffer primitives in Open vSwitch for efficient packet encapsulation and decapsulation. At the endpoints, flows are generated by a Python-based traffic generator with TCP Cubic connections.

On the switches, ECN is enabled. All the port queues share a memory of 12MB. Each queue can have up to 6.9MB memory. We set the minimum queue length threshold as 20 TCP segments and maximum threshold as 500 TCP segments. When the queue length is less than 20 TCP segments, no ECN bits are marked. When the queue length is between 20 segments and 500 segments, the ECN is marked probabilistically according to the queue length. Switches mark ECN bits on all the packets when the average queue length is larger than 500 TCP segments.

To implement the centralized approaches such as ElasticTree and CARPO, we use the ONOS [87] SDN controller to control all the software and hardware switches. There are 2 VLANs in the testbed to avoid affecting the accuracy of our measurement results. All the

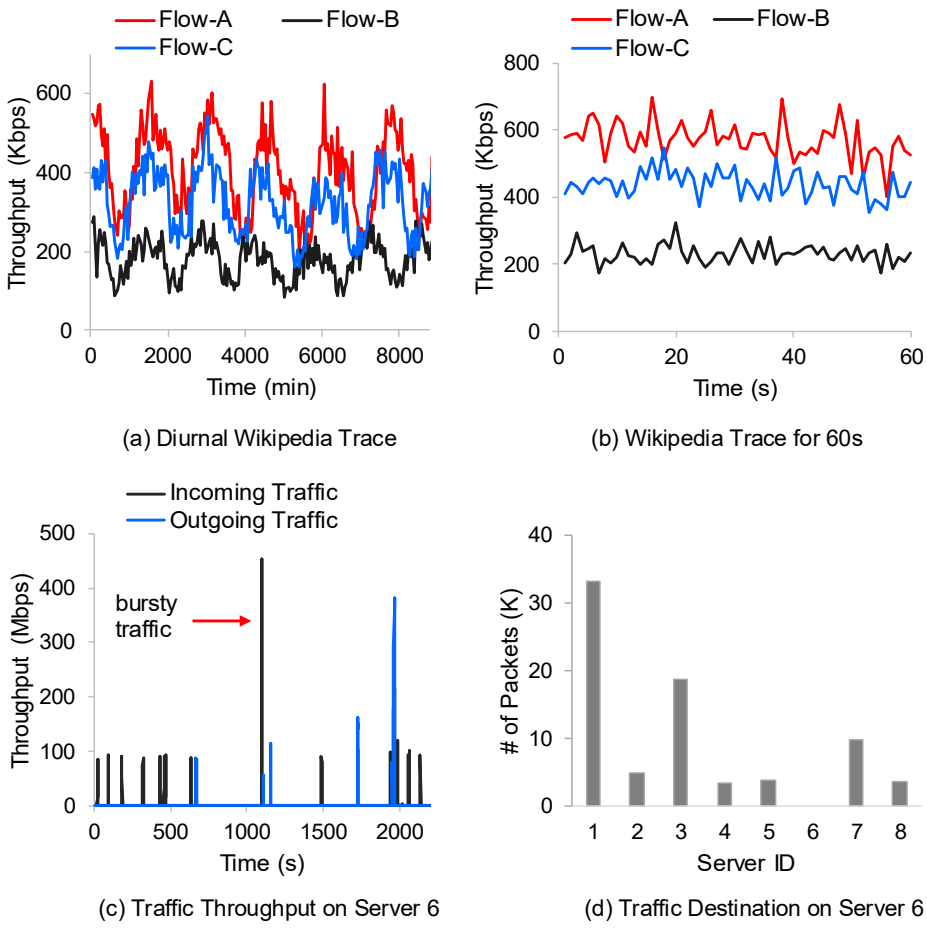


Figure 4.8: We use the Wikipedia web service (part a and b) and Facebook MapReduce (part c and d) trace to evaluate DREAM.

data traffic is routed over VLAN 20 and all the Openflow control messages are routed over VLAN 1. The Openflow protocol version we used is 1.3. The power consumption of switch is measured by using Watts Up power meter with a granularity of 0.01 watt.

We use two application workloads that are typical of those observed in current data centers: Web service like Wikipedia and MapReduce as seen in Facebook to evaluate the benefit of DREAM.

Wikipedia Trace. The Wikipedia trace [107] contains about 280 Billion HTML page requests to Wikipedia database servers. As in [113], we extract all the requests to the English Wiki pages and group the requests into different prefix folders. For example, the URL https://en.wikipedia.org/wiki/American is grouped into folder A. Thus we have 61 folders (A-Z, a-z and 1-9). We assume that each folder is served by one database server. Thus, we have 61 flows in the virtual network. Three of them are shown on Fig. 4.8 (a) and (b). Fig. 4.8 (a) is the trace for 6 days and Fig. 4.8 (b) is the results for 60 seconds. The Wikipedia trace exhibits diurnal and day-of-week pattern over the 6 days.

Facebook MapReduce Trace. We use a MapReduce trace obtained from a 2009 Hadoop log file of a Facebook cluster [19]. The SWIM [19] benchmark allows us to replay jobs from the Hadoop log file and submit these jobs to actual clusters of machines. To get the traffic matrix, we set up a 8 virtual machine cluster on a host with 64 cores and 64G memory. Each virtual machine has 2 CPU cores, 2G memory, a 140G hard disk and runs Ubuntu 14.04. In this cluster, we log the task information. Then we regenerate these tasks on our testbed using Python. The generated network traffic depends on the topology and environment we implement.

Fig. 4.8 (c) and (d) show the features of our MapReduce trace. The incoming and outgoing traffic at each slave node is negligible (about 4 Kbps) most of the time. However, as seen in Fig. 4.8 (c), there are very short time-scale spikes in the traffic. The burst in link throughput peaks for a few seconds, going up to 400Mbps. The burstiness of the trace comes from the shuffle phase in MapReduce framework. In Fig. 4.8 (d), we find that there is traffic between each pair of servers in the cluster.

4.5 Evaluation Results

In this section, we evaluate DREAM on the testbed described in last section. The Wikipedia web service [107] and the more bursty Facebook MapReduce trace [19] are used. For comparison, we also implement the baseline ECMP, as well as ElasticTree [49] and CARPO [113] designs. Baseline ECMP is implemented on a centralized ONOS [87] SDN controller. Both CARPO and ElasticTree are centralized approaches that periodically run a linear programming model to decide the routing paths for flows. We use GLPK programming language [42] to implement the linear programming model. The scheduling granularity for ElasticTree and CARPO is the single TCP flow-level.

Due to the refresh rate of hardware stats in our testbed, which is 20 seconds, we set the epoch length for ElasticTree and CARPO at 25 seconds. CARPO considers the correlation between flows and avoids placing positively-correlated flows together compared to ElasticTree. Another major difference is that CARPO uses the 90th%tile of the last epoch to predict the flow's bit rate at the next epoch, while ElasticTree uses the peak value in the last epoch. In contrast, the epoch length for DREAM is at most one RTT as the hardware switch can mark ECN on each packet. Finally, we obtain the energy savings by using a Watts up power meter with granularity of 0.01 watt. The packet drop ratio is calculated by using the TX/RX drop counters at the switches and latency is measured at the application-level.

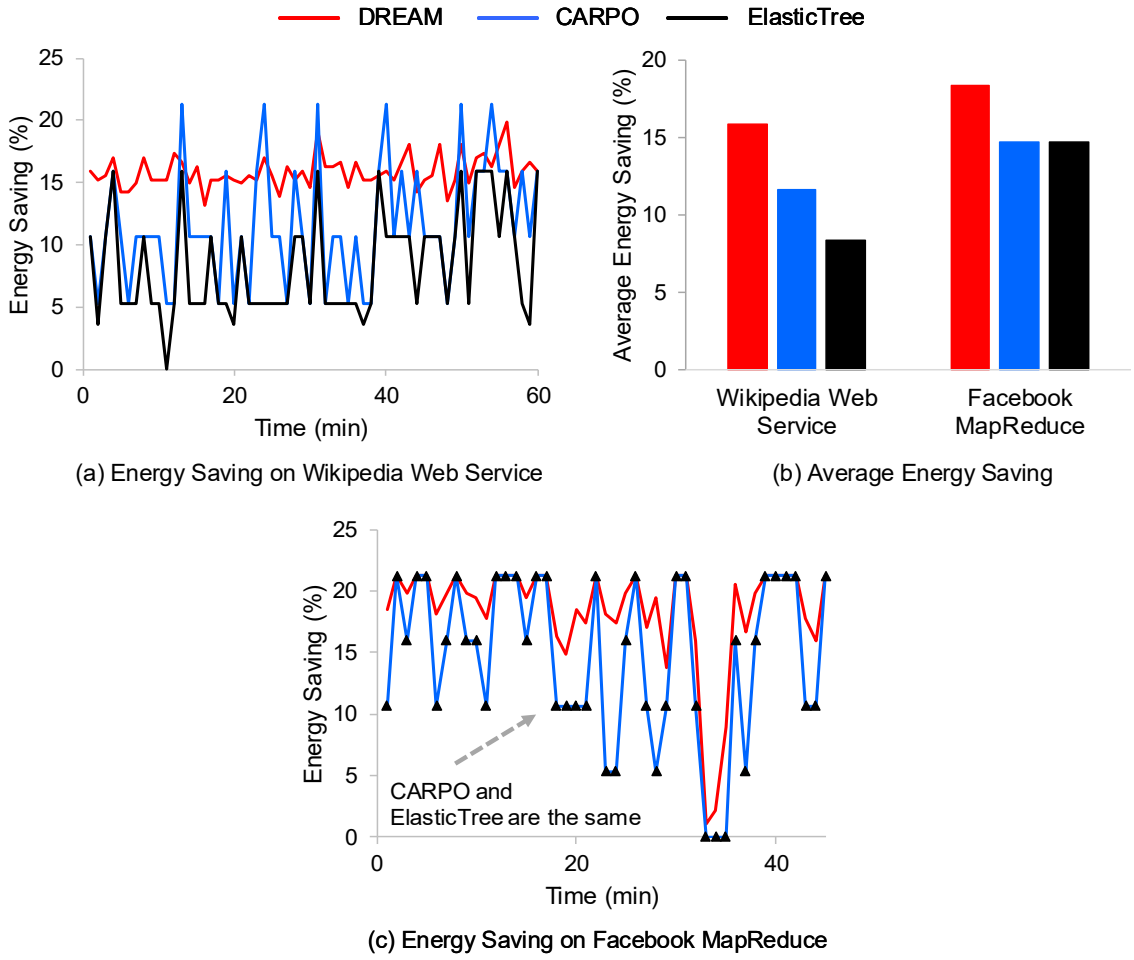


Figure 4.9: Because of good responsiveness and flowlet-level traffic scheduling, DREAM saves more DCN energy compared with CARPO and ElasticTree on both Wikipedia and Facebook trace.

4.5.1 Energy Saving

We first compare DREAM with CARPO and ElasticTree with regard to the energy saving. All the energy saving results are normalized to the energy consumed with ECMP. In the first experiment, we evaluate DREAM and the other alternatives using the Wikipedia web service workload. Each server has 8 consistent flows to other servers in the testbed. Half of them are sent to the servers within the same Top of Rack (ToR) switch. The destination of

remaining 4 flows are randomly selected outside the ToR switch. We use a traffic generator to generate the Wikipedia web service flows with TCP connections implemented on Linux. As the original Wikipedia trace has a low data rate for flows, we scale up its rate in our experiments, and utilize more than the Minimum Spanning Tree (MST) topology in the DCN.

The energy saving results on Wikipedia trace, as it varies over time (up to 60 minutes) is given on Fig. 4.9 (a) and the average results are shown on Fig. 4.9 (b). Among all the 3 frameworks, DREAM has the highest energy saving results throughout the experiment, except for 6 sample points. On average, the energy saving of DREAM is 15.9%. The improved energy saving with DREAM is due to it being more responsive to traffic variation because of its distributed design. When the traffic load ebbs, it immediately reconfigures the routing of flowlets and thus leaves more idle switches and links. On the other hand, the centralized CARPO and ElasticTree have longer 25 seconds epoch length. They miss a large number of opportunities to save energy as the traffic load decreases. The longer epoch length of the centralized approaches is constrained by the running time for the optimization and the refresh rate of TX/RX stats from the switches.

Of the alternatives, CARPO has better energy saving results compared to ElasticTree, because of its more aggressive data rate prediction and that it reserves less network bandwidth for future traffic variations. But this sacrifices congestion, reflected in packet drop rate and delay. On average, CARPO and ElasticTree save 11.6% and 8.6% DCN's energy, respectively. Only at a few time instants (note the six peaks), CARPO saves more energy compared to DREAM in Fig. 4.9 (a). Because the distributed agents in

DREAM reroute flowlets to alternative paths while we observed ECN feedback as the flow's data rate increases. On the other hand, CARPO has to wait until the end of current epoch and wakes up an alternative path in next epoch. That is why CARPO saves more energy at those six peaks. When the traffic ebbs at next epoch, CARPO reserves bandwidth based on large data rate predicted at last epoch, thus resulting in switches and links being left active and under-utilized.

The second experiment is to evaluate the schemes based on the more bursty Facebook MapReduce trace. In the MapReduce framework, the network throughput bursts only when the data shuffle phase occurs. Most of the time, traffic in the network are just the heartbeats between master and slave nodes. This is just a few Kbps in our trace. In order to have more traffic variability, we add the Wikipedia trace as background traffic in the second experiment. The energy saving results on the Facebook MapReduce trace is shown in Fig. 4.9 (c). Once again, DREAM saves the most DCN energy all the time. The average energy saving over 60 mins is 18.4%, which is 26% better than CARPO and ElasticTree. The average energy saving for CARPO and ElasticTree are similar, around 14%. Most of this is because of their poor responsiveness to bursty traffic.

4.5.2 Packet Drop

One major performance concern in traffic consolidation is that some links are heavily utilized while others are idle, to save energy. Higher link utilization increases the probability of packet drops and delay. Mitigating this requires scheduling schemes such as CARPO and ElasticTree to be able to reroute the flows quickly, especially for traffic bursts. Nevertheless, being responsive to bursty traffic is still challenging for CARPO and

ElasticTree because of their long epoch length. In DREAM, we observe that queue build-up at switches mainly contributes to the high packet drop and delay. The ECN signal in our design detects the queuing at switches early and notifies the distributed agents before the queue at switches is full. At high link utilization, DREAM can reroute the bursty traffic in a timely manner to reduce packet drop and delay. Also, path redundancy in a DCN is exploited as we transmit the flowlets of one TCP connection across multiple active paths. In this section, we compare DREAM, CARPO and ElasticTree, in terms of packet drop rates. Then, we show the application-level latency results in the next section.

The average packet drop ratios are given on Fig. 4.10 (a). On both the Wikipedia web service and Facebook MapReduce trace, the packet drop ratio of DREAM is less than 0.01%, mainly because it adapts the flowlet transmission based on queue build-up information obtained from ECN feedback. In DREAM, traffic will be moved from a path if the queue length at switches on the path exceeds the threshold. On the Wikipedia web service workload, the packet drop probability for CARPO and ElasticTree is 1.23% and 0.85%, respectively. Compared with DREAM, they are much worse due to their poor responsiveness to traffic variation. CARPO has 44% worse packet drop ratio than ElasticTree because of its more aggressive data rate prediction for a flow. It uses the 90th-percentile data rate in last epoch to predict the rate rather than the peak value used in ElasticTree. On the Facebook MapReduce trace results, the packet drop ratio for CARPO and ElasticTree is around 0.2%. Although the average packet drop ratio results for CARPO and ElasticTree are lower compared with those on the Wikipedia trace, the packet drop ratio can be as high as 6.2% when the data shuffle phase in MapReduce happens. Because there is only

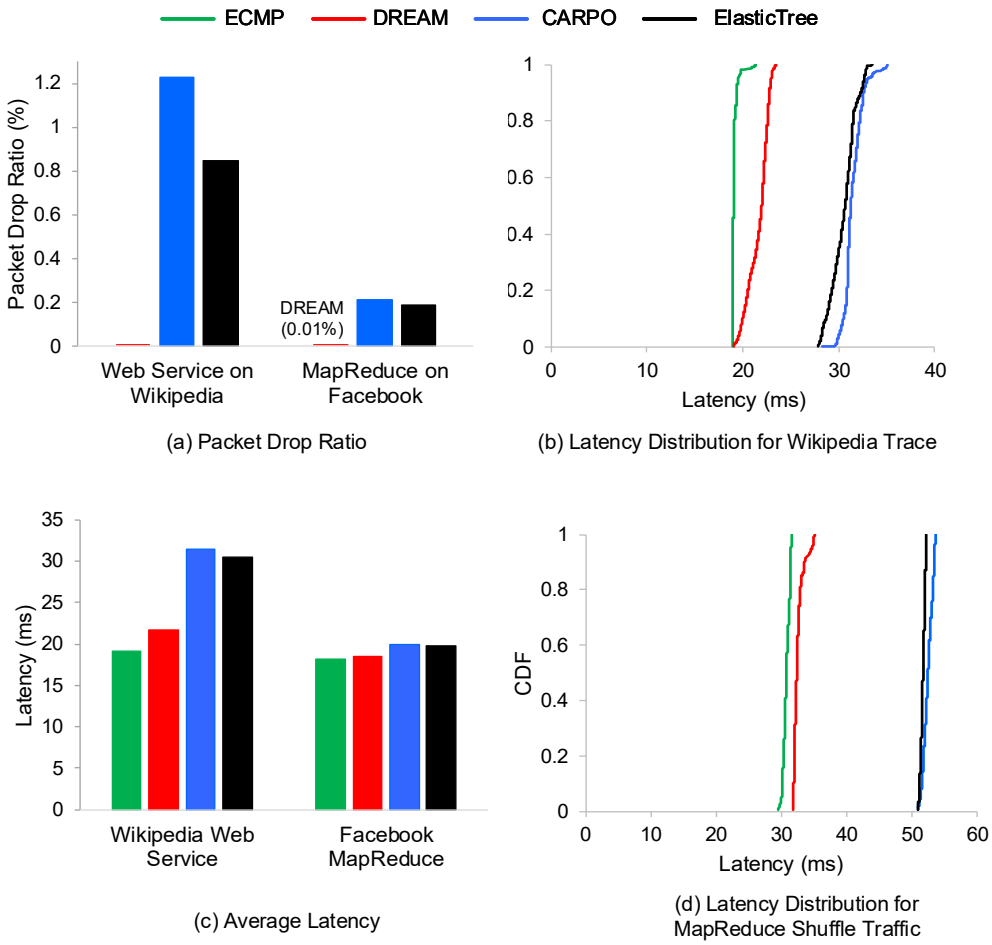


Figure 4.10: DREAM has the least average packet drop ratio on both Wikipedia web service trace and Facebook MapReduce trace. Part (b), (c) and (d) plot the application-level latency results. DREAM achieves the shortest latency among all the alternatives.

heartbeat traffic in the network most of time. Packet drops don't occur for those heartbeats. The average packet drop ratio on Facebook MapReduce trace is lower (which is somewhat misleading), but the larger number of drops during the MapReduce data shuffle phase significantly impacts the application's performance.

4.5.3 Application-Level Latency

Finally, we show the application-level latency results in Fig. 4.10 (b), (c) and (d). In the experiments, we put the time stamp in the payload of packets originating from the sender of TCP flows. On the receiver side, our program echos the inserted time stamp without any additional processing or delay. Then the application-level perceived latency can be calculated simply as $(time_{new} - timestamp_{old})/2$.

Fig. 4.10 (b) is the CDF for application-level latency on Wikipedia web service trace. The average values are given on Fig. 4.10 (c). The latency in DCN mainly comes from the queuing time at switches. The baseline ECMP has the shortest latency results, with an average of 19.12 ms. Because of traffic consolidation in DREAM, CARPO and ElasticTree, all of them have longer latency than ECMP. The average latency for ElasticTree is 30.5 ms. CARPO has larger average latency result of 31.5 ms due to its aggressiveness. CARPO reserves a much smaller headroom bandwidth. In DREAM (which has an average of 21.6 ms), we have early reaction to queue build-up by using ECN feedback. So, DREAM has 29% shorter average latency compared with ElasticTree.

The application-level latency results for the Facebook MapReduce trace are shown in Fig. 4.10 (c) and (d). As there is only a small amount of heartbeat traffic in the network most of time and all the schemes have the same active topology (i.e., the Minimum Spanning Tree), the average latency for ECMP, DREAM, CARPO and ElasticTree are almost the same in Fig. 4.10 (c). However, when the shuffle phase in Mapreduce happens, we observe the latency differences among the traffic consolidation alternatives. So, we focus on the CDF of the latency for MapReduce only during the shuffle phase in Fig. 4.10

(d). For bursty traffic, DREAM achieves at least 39% shorter latency compared to the other traffic consolidation schemes. Overall, the responsiveness and flowlet scheduling in DREAM produces much higher energy savings. The early reaction to queue build-up at switches by using ECN feedback reduces packet drops and application-level latency.

Chapter 5

Joint Server and Network Energy Optimization

5.1 Introduction

Existing server and DCN energy proportional techniques [49, 113, 125, 109, 74, 60, 24, 108] sometimes do not cooperatively reduce power as they lack the entire system view. TimeTrader [108] is one example that does borrow the network slack to provide an additional budget for computation. It only monitors the ECN bits or the RTO signal in the TCP protocol, assumes the DCN provides load balancing, and the lack of a queue build-up is treated simplistically by adding the full network latency budget to the compute slack. However, traffic consolidation invalidates this assumption and a subnet of the topology may be congested, resulting in TimeTrader becoming overly conservative in not providing any slack to the servers.

In this chapter, we address the aforementioned problem by developing a joint energy optimization framework for the entire data center, named EPRONS (i.e., Energy PROportional Network and Server). EPRONS dynamically adjusts the amount of network slack that is provided to servers by selecting different consolidation policies based on the current demand. For the servers, we find that previous DVFS-based techniques [60, 108, 69] have strict latency constraints in which the maximum frequency is selected to guarantee every request's server-side deadline. This constrains saving power. However, only the application-level SLA primarily matters to users. The network can compensate for slower server response, while still meeting application-level SLAs, guaranteeing service quality and increasing the amount of power saved. Based on this observation, we propose EPRONS-Server, a novel dynamic power management scheme that changes CPU frequency to ensure that the *average tail latency* meets the latency constraints. This enables more requests to complete service closer to their deadlines. At the DCNs, we develop a linear programming model to consolidate the traffic with latency constraints. The model chooses the best subset of active network devices and corresponding path for flows to maximize the entire data center's power saving. We also utilize heuristic algorithm to accelerate finding the solution.

To evaluate EPRONS, we implement it on the MiniNet emulator and POX [90] controller. Experimental results show that the server power saving of EPRONS-Server outperforms state-of-the-art energy proportional techniques. EPRONS can save as much as 31.25% of a data center's total power budget at low loads. In summary, this chapter makes the following contributions:

- We consolidate latency-tolerant background flows and latency-sensitive queries to save DCN power while controlling latency.
- We propose EPRONS-Server, a power management scheme on servers that dynamically adjusts frequency to enable the average tail latency of services to still meet latency constraints.
- We jointly optimize the power consumption of latency-sensitive applications by developing a new linear programming model as well as a heuristic approach.

5.2 Latency-Aware Traffic Consolidation

The primary metric targeted in current traffic consolidation techniques is the bandwidth demand. In addition to available bandwidth, another important performance metric for latency-sensitive applications in a DCN is the network latency. Previous traffic consolidation techniques [49, 113, 125, 109] only guarantee the available bandwidth for flows, but may schedule flows on highly utilized links. Thus, latency-sensitive requests or replies may miss their deadlines. On our platform, we measure the average latency of search queries under different link utilizations, and obtain the utilization-latency curve on Fig. 5.1. The results show that the network latency is well behaved at low link utilization (e.g. 20%). Moving a flow from a lightly utilized link to a link with medium utilization does not result in a substantial increase in latency. However, going beyond a utilization threshold the latency increases substantially (the 'knee' of the utilization-latency curve) due to queuing. For example, the latency grows quickly from 139 μs to 11.981 ms beyond this threshold. Thus, we observe that we can continue to consolidate the traffic to a smaller subset of the switches

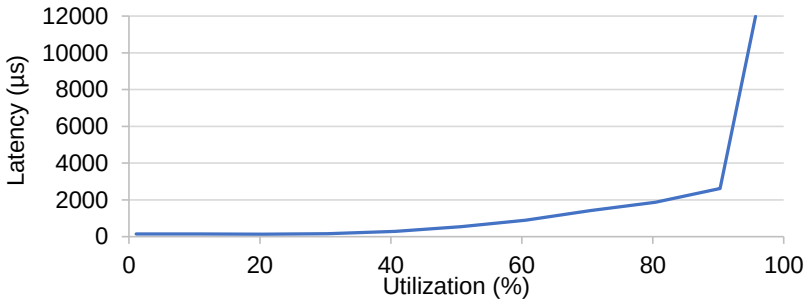
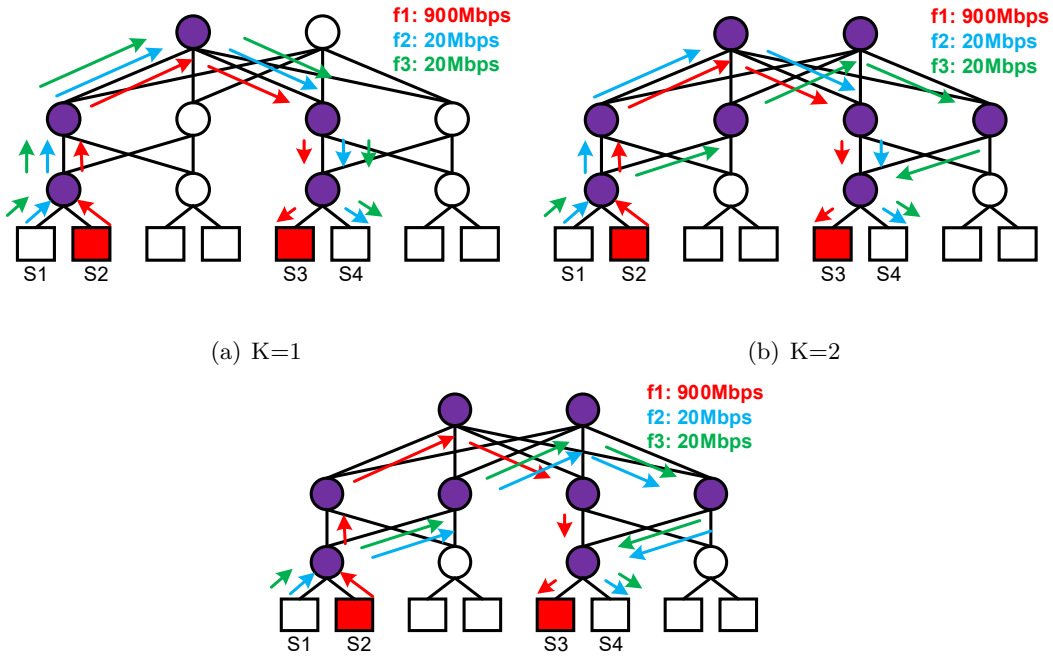


Figure 5.1: Illustration of latency knee for link utilization vs network latency.

and links in the network until we are close to the knee of the curve, thus maximizing the power savings in the DCN. Beyond this, we run the risk of missing flow deadlines, and at which point we also start taking away the benefit of adding slack to the server 'layer'.

In EPRONS-Network, we leverage an idea similar to FCTcon [126] to control the request and reply latency. We reserve additional bandwidth for requests or replies, compared to their predicted future bandwidth requirement. For example, for a request R with bandwidth requirement of B, the latency-aware traffic consolidation denotes its bandwidth requirement as $K * B$. By changing the scale factor K, we can adjust the available bandwidth on the path assigned to request R. A larger available bandwidth on the path reduces the network latency. Latency-aware traffic consolidation dynamically adjusts the scale factor K to control the network latency.

Fig. 5.2 gives an example to show how the scale factor K can affect network latency and the number of active switches. In the Fat-tree topology, all the links have a capacity of 1Gbps. The algorithm provides a 50Mbps safety margin, limiting the maximum available link bandwidth to 950Mbps. There are 3 flows in the DCN. The red flow with 900Mbps data rate is a latency-tolerant 'elephant' flow, while the two latency-sensitive flows



(a) K=1

(b) K=2

(c) K=3

Figure 5.2: Fat-tree topology, 1Gbps link capacity and 50Mbps safety margin. A latency-tolerant flow (red) and 2 latency-sensitive flows (green and blue).

(blue and green) have a requirement of 20 Mbps each. In Fig. 5.2(a), the 2 latency-sensitive flows share the same path with the elephant flow when we set the scale factor K to 1. With $K = 1$, the number of active switches is at a minimum and the DCN consumes the least amount of power. But the 2 latency-sensitive flows are likely to suffer a higher latency because many of the inter-switch links are almost fully utilized. In Fig. 5.2(b), we change the scale factor K to 2. This results in moving either the green or blue flow to a new path because of bandwidth constraints. More switches have to be turned on, but we decrease the latency for flows on those links. When we further increase the scale factor K to 3 in Fig. 5.2(c), both the blue flow and the green flow are scheduled on different paths, thus further reducing the network latency.

5.3 Dynamic Power Management on Servers

A common technique to save energy in latency-sensitive applications is to exploit latency slack by slowing down request processing so that the target tail latency is just satisfied [60, 69, 68, 24]. Intuitively, the processor should run at the minimum frequency that satisfies the request's deadline. However, this is difficult as frequency selection also affects the latency of subsequent requests due to request queuing. In TimeTrader [108], frequency settings are periodically updated based on the monitored tail latency. While effective with smooth diurnal traffic variation, these prior techniques result in unacceptable latency constraint violations as they lack responsiveness, which is crucial with fast-varying bursty traffic patterns that is common in latency-sensitive applications[54, 60, 24].

To address this problem, [60, 24] use per-request latency distribution to compute the slowdown/delay for each request, and find the configuration that satisfies *all* queued requests. The frequency setting is then determined by the request with the least latency slack. While satisfying latency constraint, this conservative frequency selection does not fully exploit the energy saving opportunities of latency slack, since the requests other than the limiting request will have their tail latency shorter than the constraint.

These prior techniques all assume a fixed request deadline [60, 69, 68, 24]. However, when coordinating server power management with network power management, the request deadline is extended with a random value taking advantage of additional network slack, which varies per request. Therefore, the latency variation behavior of the DCN should be considered. In this section, we design the EPRONS-Server under a more generalized assumption, with variable request deadlines. EPRON-Server is a slack variation-aware

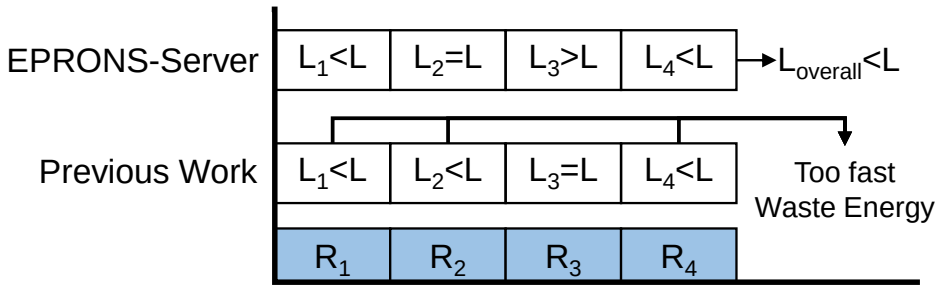


Figure 5.3: High level idea of EPRONS-Server.

power management scheme that determines the request processing speed based on the per-request server and network slack.

5.3.1 Details on EPRONS-Server

In prior work, the frequency selected is based on the highest frequency required to process a queued request just-in-time to meet its deadline. This results in a single request meeting tail latency constraints just-in-time, but with all other requests finishing before the deadline. This is illustrated in Fig. 5.3. There are 4 requests in the queue ($R1$, $R2$, $R3$ and $R4$). The latency constraint is given as L.

L_i is the tail latency of request R_i under the frequency determined by the scheduler. In the previous work, the scheduler determines the frequency for each request so that L_i is smaller than L, as a result, all the queued requests can meet their deadlines. Due to this policy, power savings become worse with the latency slack variation introduced by network power management. To account for latency slack variation, the main insight behind EPRONS-Server is that we can *slow down request processing so that* ***some queued requests*** *will exceed the tail latency constraint.* We only need the *overall tail latency of the*

queued requests to satisfy the constraint. One motivation behind sacrificing some queued requests' miss rate for better energy saving is that the 'additional delay' due to lower selected frequency can be compensated by the slack for the reply. However, it is not factored in prior work [60, 108, 69]. Thus, the 'additional delay' for some queued requests on servers has limited impact on the performance and brings better energy saving.

The challenge in designing EPRONS-Server lies in determining the tail latency of each request under different frequencies. To account for the queuing effect and short-term variation, EPRONS-Server uses a performance model based on the request's probability density function (PDF), similar to [60, 24]. In our experiments, the PDF is obtained through measuring the service time distribution of Apache Solr search engine [40]. When we dynamically change the frequency setting, the request service time can be calculated accordingly. Based on the model, we derive the *violation possibility* (VP) for each frequency setting, and the new frequency is then determined based on the average VP for all the queued requests at every request arrival and departure instance. The goal is to select an average VP that will not violate the tail latency constraint.

To illustrate this, take the following example. Assume we have three requests ($R1$, $R2$, $R3$) in the queue. In order for the requests to finish just-in-time, they would require to run at 1GHz, 1.2GHz, and 1.1GHz. Prior works would set the processor at 1.2GHz for $R2$, ensuring that the latency constraint would be met. If the SLA is set at the 95th %tile latency, it allows for a violation probability of 5%. Even if we set the processor at 1.2GHz, $R2$ has a 5% chance of exceeding the deadline, while $R1$ and $R3$ would have less due to running at a higher frequency. This results in some lost energy savings potential for $R1$ and

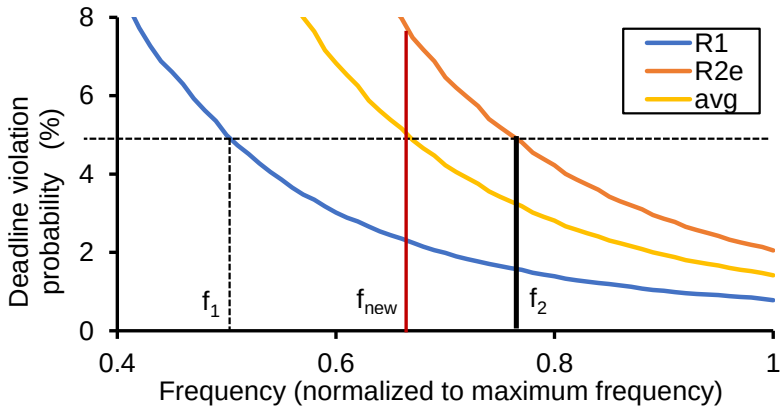


Figure 5.4: Energy saving opportunities with *average* tail latency.

$R3$. The goal of EPRONS-Server is to select a frequency where the violation probability of all three requests combined is 5%. In order to achieve this, we simply need the *average VP* of all three queued requests to be 5%.

Fig. 5.4 gives an example of how EPRONS-Server chooses the operating frequency. Assume, at time $t = 0$, the core finishes a request and is ready for the next. Requests $R1$ and $R2$ arrived previously and are queued in the buffer, with deadline $D1$ and $D2$, respectively. In previous work, finding the processing frequency for $R1$ is equivalent to finding the *maximum frequency* that satisfies both request $R1$ with start time $t = 0$ and deadline $D1$, and a request $R2e = R1 + R2$ with start time $t = 0$ and deadline $D2$. We call $R2e$ the *equivalent request* of $R2$, which is defined as the convolution of a given request, and all requests ahead of it in the queue. $R2e$ is a convolution of the amount of work required for both $R1$ and $R2$, as $R2$ can only complete after $R1$ completes.

Fig. 5.4 plots the deadline violation probability for request $R1$ and $R2e$ under different operating frequency (X-axis). The horizontal dotted line represents deadline miss rate constraint of 5%. The frequency f_1 and f_2 will satisfy the deadlines for request $R1$ and

$R2e$, respectively. Since $f2$ is the minimum frequency that satisfies the miss rate constraint for both $R1$ and $R2e$, the core will process $R1$ at frequency $f2$. While guaranteeing meeting the deadline miss rate constraint, this conservative frequency selection leads to unnecessarily high request processing speed for $R1$. As we can see, under frequency $f2$, $R1$ will have it's VP (1.8%) far lower than the required miss rate (5%), which results in inefficient energy consumption.

In EPRONS-Server, instead of choosing the maximum frequency required for all requests, we select the frequency based on the *average VP*. As plotted in the Fig. 5.4, the average VP falls between the VP of $R1$ and $R2e$. Thus, we can run at frequency f_{new}, which significantly reduces the frequency and saves more energy. Although it leads to the miss rate constraint violation of $R2e$, the high miss rate of $R2e$ is compensated by the low miss rate for $R1$, hence the overall miss rate constraint is still satisfied.

5.3.2 Violation probability

The following model is used for determining the processing frequency at the request departure instance. For a request in the queue, the amount of work that can be done under frequency f until the deadline D is

$$\omega(D) = f \times (D - T_{start}) \tag{5.1}$$

where T_{start} is the time the request starts to be processed. The violation probability distribution for $\omega(D)$ can be found by computing the complementary cumulative distribution function (CCDF) from the *request's equivalent distribution*. Fig. 5.5 shows the violation probability distribution for three equivalent requests: $R1e$, $R2e$, and $R3e$. Find-

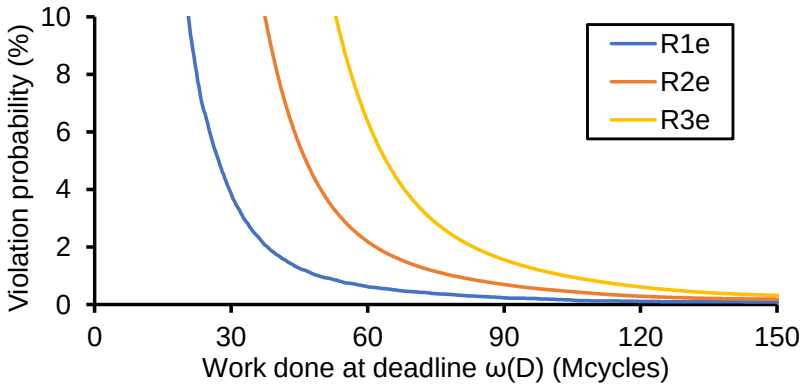


Figure 5.5: Violation probability of three requests under different $\omega(D)$.

ing the VP is simply finding the corresponding "y" on a line given the "x". By combining the request's equivalent distribution with equation (5.1), we can get the VP for different frequencies and deadlines.

The above model is used for determining the processing frequency at the request departure instance. It can also be applied at the request arrival instance with a minor modification. Consider a situation when a request arrives at time $t = A$ and finds that the core is processing a request, $R0$. It is equivalent to the scheme that the core just starts processing a request $R0e$ with the distribution equal to the distribution of the *work left* of request $R0$, during time $t = 0 \sim A$. The equivalent distribution of the queued requests will then be $R1e = R0e + R1$, $R2e = R0e + R1 + R2$ and so on.

5.3.3 Reducing computation overhead

The most computational intensive portion of our proposed scheme lies in finding the *equivalent distribution* for the queued requests. For the nth queued request, it needs $n - 1$ convolution operations. With the use of Fast Fourier Transform (FFT), computing

one convolution requires 20μs in our machine. To further reduce the computation overhead, we take the assumption that the request distribution will be similar within a period of time. Thus, the equivalent distributions can be reused once it is computed. However, at the request arrival instance, since the conditional probability is unknown beforehand, we have to perform n convolutions. This overhead is considered while determining the frequency by replacing D in equation (5.1) with $D' = D - overhead$.

Once we have the equivalent distributions, the time it takes to determine the operating frequency is shortened by applying binary search on the average VP. In our experiments, it takes less than 30μs and can be safely ignored, since the request processing time usually falls in the millisecond range.

5.4 Joint Power Management

5.4.1 Latency and Power Analysis

In joint server-network power management, the scale factor K is the key to controlling the entire data center's power consumption while satisfying all the latency constraints. It can be utilized to trade-off the DCN power and DCN latency. As mentioned in section 5.2, the network latency will decrease if the flow is routed along less utilized links. This can be achieved by enlarging the scale factor K and thus allocating more free bandwidth on the assigned path. Accordingly, the larger K will result in higher DCN power consumption because additional links and switches are activated. In real deployments, it would be hard to predict network latency based on current network conditions, as it is highly workload dependent. In EPRONS, we use a portion of the application queries to train our model and

then use this model, assuming that the workload features don't change significantly for the same application. Then, we measure the average tail latency of search queries for different scale factors K and use this information for determining network resource allocation.

On servers, power consumption is affected by server utilization and tail latency constraints (i.e., server time budget). Prior works [60, 69, 74] assume that every request of latency-sensitive applications has the same fixed tail latency constraint. If the server layer borrows some slack from the network layer and use it to increase the server time budget, we can make the processing on servers slower and thus save more power, assuming that the server utilization doesn't change. Because the scale factor K can control network latency/slack, it will indirectly affect the server processing time budget and thus the server power consumption. Similarly, we measure the server power consumption for different utilizations and tail latency constraints that may then be used to parameterize our model.

5.4.2 Optimization Model

The design motivation behind EPRONS is to jointly optimize the entire data center's power consumption without violating both network and server tail latency constraints. We formalize this optimization problem as a linear programming model, in which the objective function minimizes the total system power by searching for the optimal scale factor K. Our model is as follows:

Objective function:

$$\text{minimize} \quad \sum_{(u,v)\in E} X_{u,v} * l(u,v) + \sum_{u\in V} Y_u * s(u) + N * P_{server} \tag{5.2}$$

Constraints:

$$1 \leq K \leq K^{max} \tag{5.3}$$

$$\forall (u,v) \in E, \sum_{i=1}^{j} f_i(u,v) \leq X_{u,v} * c(u,v) \tag{5.4}$$

$$\forall (u,v) \in E, f_i(u,v) = -f_i(v,u) \tag{5.5}$$

$$\forall i, \forall u \in V, \sum_{h \in H_u} f_i(u,h) = \begin{cases} K * d_i & u = s_i \\ -K * d_i & u = t_i \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

$$\forall u \in V, \forall h \in H_u, X_{u,h} = X_{h,u} \leq Y_u \tag{5.7}$$

$$\forall u \in V, Y_u \leq \sum_{h \in H_u} X_{u,h} \tag{5.8}$$

$$\forall i, \forall (u,v) \in E, f_i(u,v) = K * d_i * Z_i(u,v) \tag{5.9}$$

Given the network topology $G =< V, E >$, we use X and Y to indicate the ON or OFF states for the links and switches respectively. $l(u,v)$ is the power consumption of link (u,v) and $s(u)$ is the power consumption of switch u. N is the total number of servers and P_{server} is the power consumption of each individual server. The objective function minimizes aggregate power consumption of all the switches and servers. Equations (5.4), (5.5), (5.6) are the well-known constraints in network flow problems. The only difference is that we use a scale factor K to multiply the original flow bandwidth demand. $c(u,v)$ is the capacity of link (u,v) and $f_i(u,v)$ is the size of flow on link (u,v) for flow i. Every flow has source s_i, destination t_i and bandwidth demand d_i. Equations (5.7) and (5.8) guarantee that the switch will go to sleep state if all the directly connected links are off. $Z_i(u,v)$

means that if flow i will use link (u, v). With equation (5.9), we avoid splitting flows to avoid packet reordering. Our optimization model is independent of the network topology. Similar to [49, 113, 109, 127], we run the optimization every 10 mins. This is a reasonable time in practice, but can be changed depending on the network's characteristics. By solving this linear programming model with CPLEX, we can find the best flow assignments and active switches. This model is only used to find the best flow assignments. EPRONS-Server will utilize the actual monitored network slack to extend the server time budget.

Although the linear programming model can be solved in polynomial time [49], it takes a long time to find the optimal solution for a large DCN topology, unacceptable with bursty data center traffic. For example, the computation time of the linear programming model can be more than 42 min. on our platform, with 3000 flows in a 4-ary Fat-tree topology. In real deployment, we design the heuristic algorithm (similar to the greedy bin-packing algorithm in [49]) to accelerate the latency-aware traffic consolidation. In the current design, we ignore the switch ON/OFF transition overheads because we use a software switch. However, our measurement on a HPE switch show that the power-on time is about 72.52 sec. We can avoid the transition overheads by having 'backup' paths, as described in [109] or a novel hardware design with sleep states [49].

5.4.3 Framework of EPRONS

Fig. 5.6 gives an overview of our framework. The system includes two distinct parts: the EPRONS-Server module on every server and the Optimizer (EPRONS-Network) in the centralized SDN controller. On servers, we assume that each server has the same

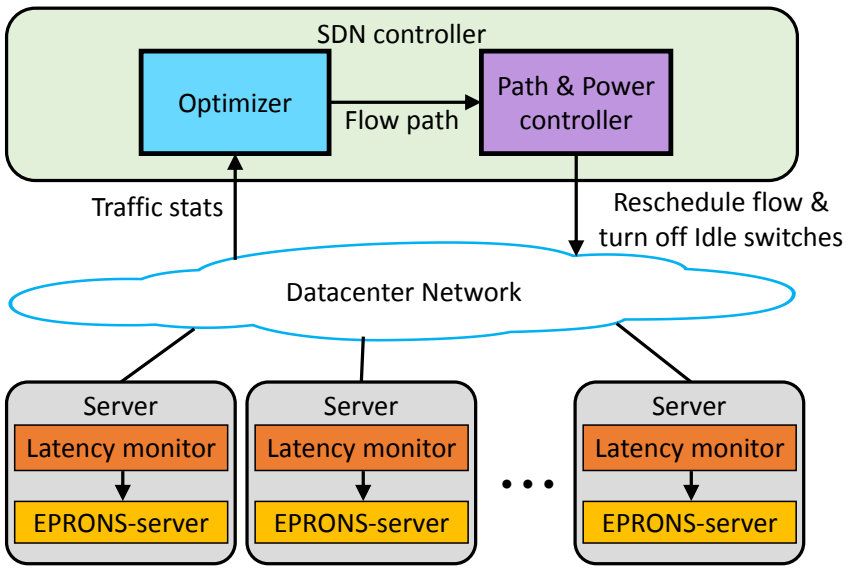


Figure 5.6: Framework overview of EPRONS.

service time distribution. The latency monitor module measures each request's network latency. To be more conservative, we only use the request slack. The EPRONS-Server module adds the different network slack of each search request to its compute budget. Then we dynamically decide the CPU core frequency to guarantee that the *average tail latency* of requests to meet the tail latency constraints. In the centralized SDN controller, we gather both the utilization at servers and the traffic matrix inside the DCNs. This information is periodically fed to the Optimizer to find the best active subgraph of the topology and corresponding flow assignments. The Path & Power controller is responsible for inserting new forwarding rules and turning off idle switches.

5.5 Implementation

MiniNet. We use a 64-core machine with 2GHz CPUs and 64G memory to build a virtual network using the MiniNet emulator that includes virtual hosts, virtual network interfaces, software switches (Open vSwitch) and software SDN controller. Each host in MiniNet is a shell process that runs the actual program. In our implementation, we create a 4-ary Fat-tree topology with 16 servers.

POX. We use POX [90] as our SDN controller. The POX controller fetches flow statistics and link utilization every 2s with an openflow message. Then, the optimizer in POX runs periodically (every 10 mins.) to find flow paths which minimizes the entire data center's power consumption with latency constraints.

Server. The performance of virtual host is limited in single machine MiniNet setup. We attempted to run the Apache Solr search engine inside Containers [40]. But it was difficult to launch 16 Docker Containers and 20 Open vSwitch instances on a single machine because of the limited number of cores and memory. In the future work, the MiniNet network could be extended to a cluster of servers.

To mimic the partition-aggregation behavior, we built a search engine simulator in the MiniNet. Instead of running real query processing, our simulator generates the service time of a query based on the service time distribution, and uses an empirical model to scale the service time for different operating frequencies. We acquire the servers' service time distribution through real machine experiments with the open-source Xapian distributed search engine [119]. We built indexes of the Wikipedia's English XML export [107], which contains all the Wikipedia articles/pages. These document indexes are then partitioned and

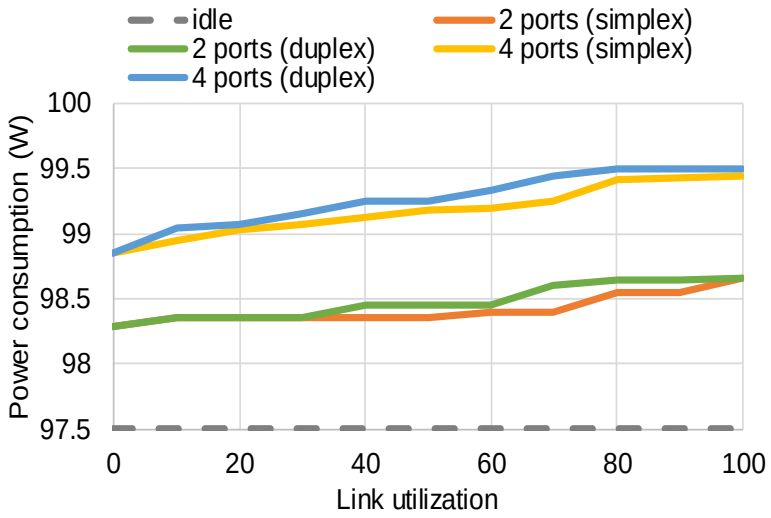


Figure 5.7: Switch (HPE E3800 J9574A) power for varying link utilizations.

distributed across 16 Xapian index nodes. We randomly generate 100K search queries, run and log their processing time on the ISNs. In our simulator, we randomly choose a server to be the aggregator, while the other 15 servers will then be the ISNs for each user query. The aggregator will broadcast sub-queries to all ISNs. Each ISN serves the sub-queries and executes EPRONS-Server to save server power.

Power Evaluation. To acquire the power consumption, our server simulator records the processing time for each frequency setting. The average power consumption is calculated based on the time and power consumption under each frequency setting. We set the frequency range between 1.2-2.7GHz in 100 MHz steps. We measure the power consumption under different frequency settings on a 12-core Xeon E5-2997 v2 CPU. The measured power consumption under maximum and minimum frequency setting is 4.4W and 1.4W for a CPU core, respectively. In our simulator, each server has a CPU with 12 cores. Moreover, we also consider the power consumption of other components (motherboard,

main memory, etc.) of the system. Based on the ratio between dynamic power and static power for a Huawei XH320 V2 server, the static power is set to be 20W.

To investigate how link utilization affects switch power, we measure the power consumption of a HPE switch (E3800 J9574A) as we gradually increase the link utilization. The switch idle power is 97.5W. Fig. 5.7 shows that the increased power consumption (0.59W) is negligible when we change the link utilization from 0 to 100%, independent of whether we activate 2 or 4 ports. That is only 0.6% of the idle power. So, we assume the switch power remains the same when the utilized network bandwidth changes. Since the power consumption doesn't increase, there is no increase in the temperature of the switch. Both of the above observations are in line with prior work [49, 113, 109]. For network power consumption, we use the measurement result of a 4-port switch in [121]. The power consumption of the active switch is 36W. These power models are selected to be a closer match to our implementation platform.

5.6 Results

In EPRONS, we optimize the entire data center's power consumption while satisfying the latency constraints. Although there are many parameters in our model, the scale factor K is the only parameter to be tuned for optimization. In this section, we investigate the interplay of all these parameters and evaluate the power saving of EPRONS on different configurations. Then, we compare EPRONS with other energy-proportional techniques.

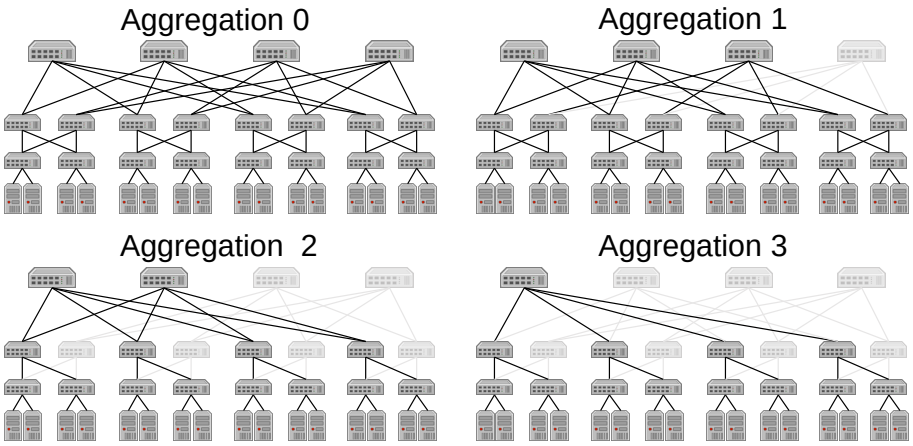
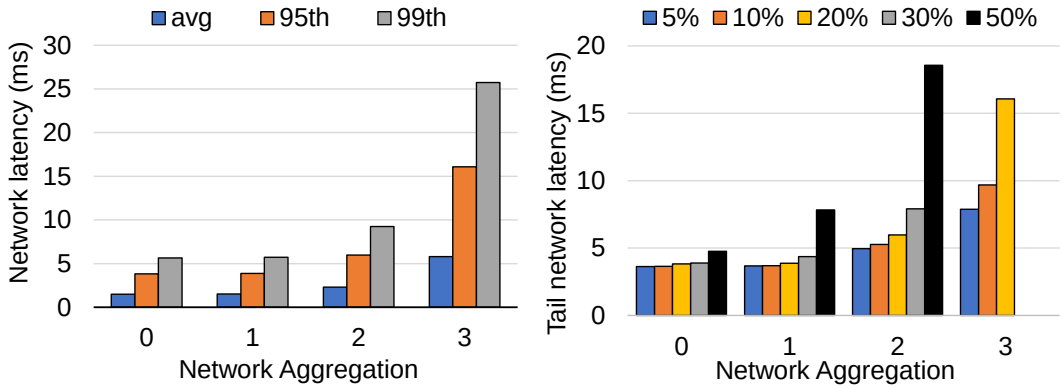


Figure 5.8: 4 different network topologies after consolidation. The greyed out switches and lines represent the deactivated switches and links, respectively.



(a) 20% background traffic

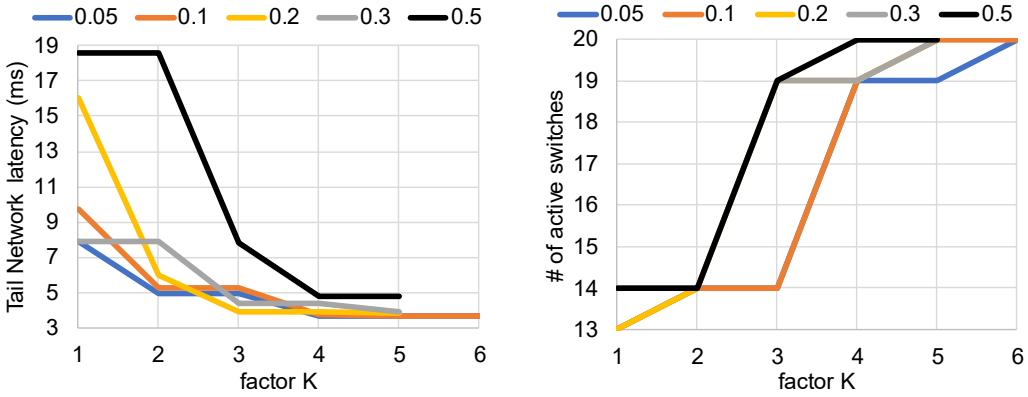
(b) 95th tail latency

Figure 5.9: Network latency under different degrees of aggregation.

5.6.1 Network Power Management

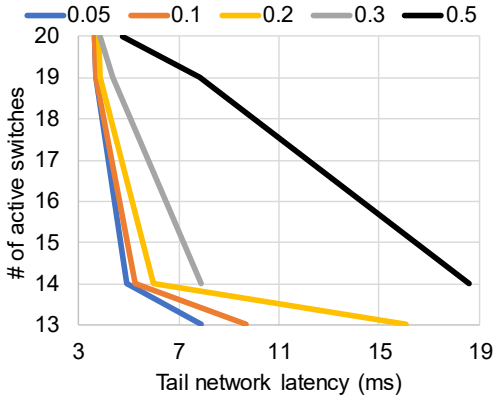
In DCNs, the traffic aggregation policy has a significant impact on the network latency. Fig. 5.8 shows all the possible aggregation policies in the 4-ary Fat-tree topology. From Aggregation 0 to Aggregation 3, we gradually turn off the core-level switches and the corresponding aggregation-level switches.

The network latencies of search queries are shown in Fig. 5.9. Apart from the



(a) network tail latency

(b) # of active switches



(c) network tail latency vs # of active switches

Figure 5.10: Network sensitivity results.

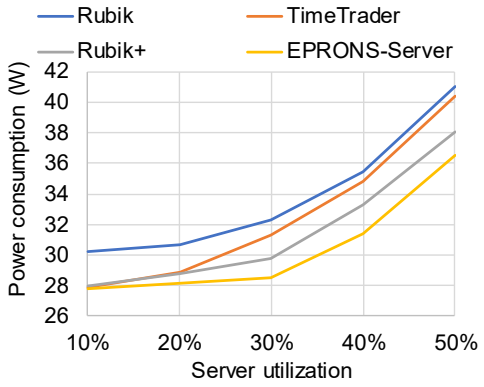
aggregation policy, the background traffic has an impact on the network latency as well. So, we first fix the network background traffic at 20% of link capacity in Fig. 5.9(a). The network latency (especially 99th %tile) increases considerably when we consolidate traffic to a smaller portion of the topology. From aggregation 0 to aggregation 3, the 99th %tile latency increases from 5.64ms. to 25.74ms. Fig. 5.9(b) plots the impact of the aggregation policy on the 95th %tile tail latency for various background traffic. The 95th %tile tail latency follows a similar increasing trend under different network utilizations.

In Fig. 5.10, we show that the scale factor K is a good parameter to trade-off between network tail latency and network power consumption. Fig. 5.10(a) and Fig. 5.10(b) compare the variance of tail network latency and # of active switches when we increase the scale factor K. Each line represents one possible background traffic, in terms of link utilization. Larger K results in smaller network tail latency. Accordingly, more switches are activated. For example, the network tail latency with 50% background traffic (black line) decreases to 4.75ms if the K is set to 4. The reason is shown on Fig. 5.10(b). We observe that 6 more switches are turned on. Fig. 5.10(c) plots the # of active switches vs. the tail network latency. Each point on the line represents one value of K. The optimal scale factor K is the point that is closest to origin (0, 0). We observe that K can effectively trade-off between # of active switches and tail network latency.

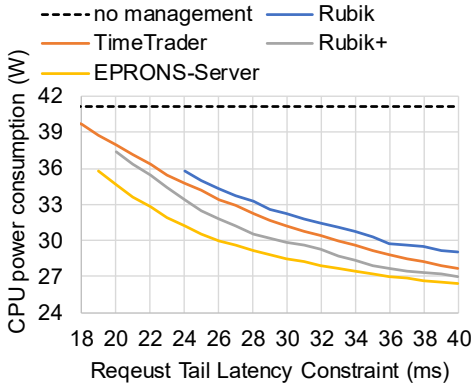
5.6.2 Server Power Management

Fig. 5.11 gives the power consumption with only the EPRONS-Server. In these experiments, we do not apply any network power management, and the background traffic is set to achieve 20% network utilization [108, 49]. To show the effectiveness of EPRONS-Server, we also implement and compare our results with two state-of-the-art server power management techniques: Rubik [60] and TimeTrader [108]. Since Rubik does not consider network latency/slack, for a fair comparison, we also implement a network aware version of Rubik (Rubik+), which monitors and uses the network slack.

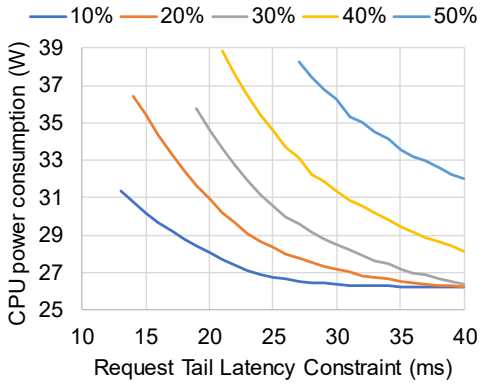
Fig. 5.11(a) shows the CPU power consumption for different server utilizations. In these experiments, we set the request's tail latency constraint to be 30ms (25ms server



(a) Server utilization vs Power



(b) Request tail latency constraint vs Power



(c) (utilization, latency constraint) vs Power for EPRONS-Server

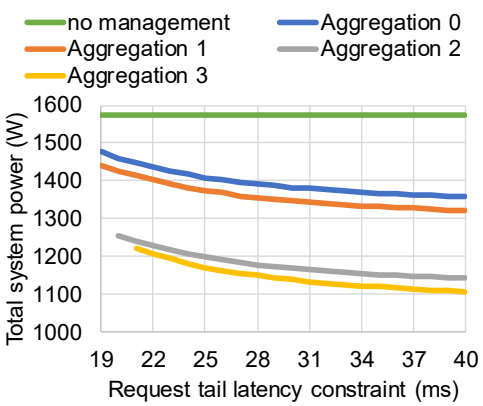
Figure 5.11: Server sensitivity results.

budget and 5ms network budget). We can see that Rubik consumes the most power across all load levels. Since Rubik does not consider network latency, it does not fully exploit the slack in the request's latency. Moreover, except at very low loads (10%), Rubik+ and EPRONS-Server consistently outperform TimeTrader. TimeTrader uses feedback to select the CPU frequency to meet the request tail latency constraint. The simple control algorithm in TimeTrader changes the CPU frequency every 5 seconds [108], and is unable to fully exploit

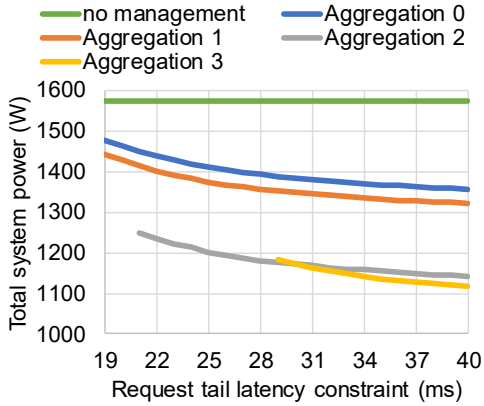
power saving opportunities. But Rubik+ and EPRONS-Server adjust the CPU frequency based on a statistical performance model at a per-request granularity. Finally, EPRONS-Server outperforms all other schemes across the entire utilization range. Compared with Rubik+, EPRONS-Server reorders requests based on their deadlines and uses the average VP, instead of the maximum, to determine the request processing frequency. By doing so, EPRONS-Server can better utilize the network aware slack in compute budget and achieve lower power consumption.

In Fig. 5.11(b), we vary the server budget from 5 to 35ms while fixing the network latency budget at 5ms. The search load is set so as to have the server utilization at 30% [74]. First, no scheme is able to meet a request's tail latency constraint smaller than 18ms. Second, in the range of 18-19 ms, TimeTrader is able to meet the request tail latency constraint and save power. But, for a request tail latency constraint at 19 ms or greater than 18ms, EPRONS-Server consistently outperforms all other schemes because of its efficient utilization of the request slack.

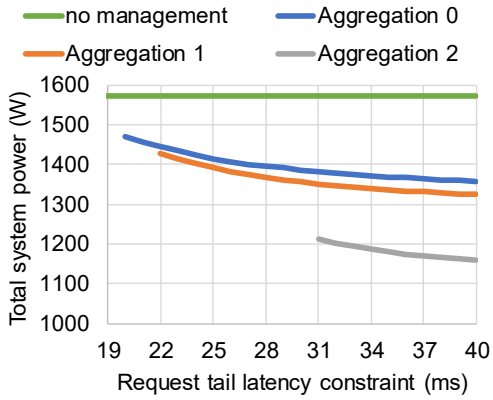
Fig. 5.11(c) shows the power consumption of EPRONS-Server with different latency constraints and search loads. Each line in the figure represents a server utilization ranging from 10% to 50%. As we see in the figure, the CPU power consumption decreases significantly for all server utilization levels as the request latency constraint increases at the smaller values. This justifies the approach of EPRONS to give the additional slack from network budget to the server to achieve a significant reduction of CPU power consumption.



(a) low background traffic (1%)



(b) medium background traffic (20%)



(c) high background traffic (50%)

Figure 5.12: Total system power under different degree of network consolidations. This result is scaled based on the result of our MiniNet experiments with 30% server utilization.

5.6.3 Joint Power Management

We now show the compelling benefit of combining appropriately the optimization of server and network power while meeting application latency constraints. In one experiment (Fig. 5.12), the server utilization is set at 30% [74]. We then change parameters such as request tail latency constraint, network aggregation policy and background traffic to see their impact on total system power consumption. When the background traffic is negligible (1%, Fig. 5.12(a)), the 4 different aggregation policies can nearly meet all the tail latency

constraints. Aggregation 3 consumes the least total system power. As we increase the tail latency constraint, the data center consumes less power, particularly because we can have longer idle times at servers. We then add more background traffic (20%), as seen in Fig. 5.12(b). The total system power has a similar trend, except that aggregation 3 cannot support a tail latency constraint less than 29ms.

Between a tail latency constraint of 29ms and 31ms, one interesting result is that EPRONS can save more power if we deliberately turn on a switch. From aggregation 3 to aggregation 2, more switches are activated and thus we have larger network slack. The benefits of this network slack is amplified in our EPRONS-Server technique which can more than compensate for the increased power consumption in the network. The goal of EPRONS is to properly trade-off network slack and server slack to minimize total system power, not just simply combining them. Finally, as we increase the background traffic to 50%, Fig. 5.12(c) shows that aggregation 3 is not feasible, and even aggregation 2 requires a latency constraint greater than 31 ms.

In the next experiment, we use the Wikipedia trace [107] to evaluate EPRONS. Both the search load and background traffic span one 24 hour period and follow a diurnal pattern. Prior traffic consolidation techniques [49, 113, 109] don't guarantee network latency constraints, and energy proportional server techniques [60, 74, 69] don't save DCN power. We compare EPRONS with TimeTrader [108], which is a cross-layer energy-proportional technique and meets both network and server latency constraints.

Fig. 5.13 gives the power saving results. On average, the total system power saving of EPRONS is a factor of *more than 2* better than TimeTrader. EPRONS saves

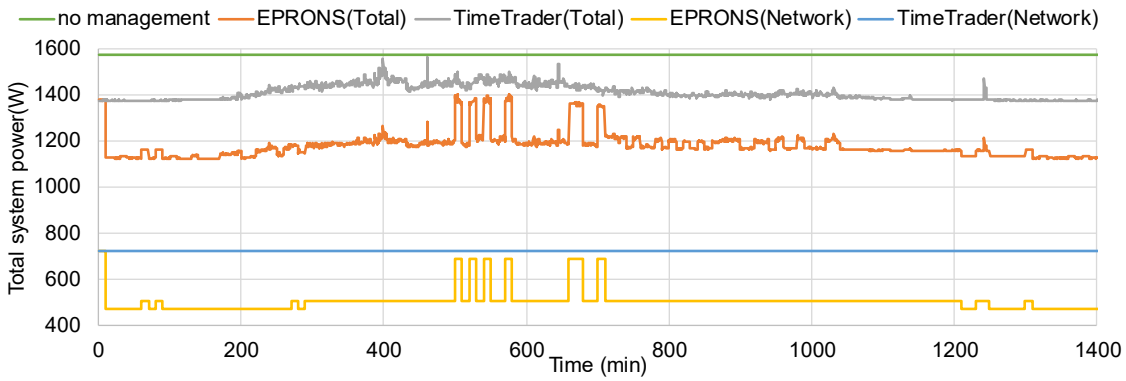


Figure 5.13: Total system power consumption with the varying search load and background traffic.

as much as 25% of the total system power, compared with only 8% in TimeTrader. Since EPRONS-Server is a more responsive per-request technique compared with the feedback-based TimeTrader, even our server-side power saving outperforms TimeTrader by 2%. The detailed total system power over the 24 hour interval is shown in Fig. 5.13, with a 1 min. granularity. We observe that the DCN power consumption of EPRONS follows the diurnal pattern. On the other hand, TimeTrader doesn't save any DCN power. For the total data center power consumption, EPRONS can save up to a maximum of 31.25%, measured over one minute intervals, of total system power. This occurs during the night, because of the lower workload intensity. The maximum power saving in TimeTrader is only 12.5%. We conclude that jointly optimizing power consumption on both DCN and servers (EPRONS) can make the entire data center more energy efficient.

Chapter 6

Gemini: Neural Network based Two-Step DVFS for ISNs

6.1 Introduction

Chapter 5 indicates that existing DVFS schemes for search applications exploit the latency slack on ISNs by slowing down request processing and finishing the job just-in-time. They seek to estimate each request's service time and then properly re-configure the current power management setup. The main challenge lies in the fact that search requests' latencies have both short and long term variation and a request's computation requirement (i.e., total CPU cycles) cannot be predicted accurately [60, 48]. To overcome the above challenge, we present Gemini, a fine grained two-step DVFS scheme with accurate per-query service time prediction. Our major goal is to properly schedule the various clock domains in DVFS [118] for optimal power saving and achieving a very limited deadline violation rate.

In the first step upon a request arrival, Gemini selects an initial CPU frequency according to the predicted service time based on a Neural Network (NN) model. In the second step, depending on the query progress at a particular time, Gemini judiciously boosts the CPU frequency, to catch up with the request's deadline, if the request processing is lagging. The boosted frequency is set to the maximum core frequency, but the boosting time must be determined accurately. In our model, this time is calculated based on a second NN error predictor for the latency estimation. Based on these two separate NN predictors, Gemini automatically adjusts a request's initial frequency and boosting time to accommodate the query specific variations.

Initial estimation of the frequency is very important to satisfy the deadline constraint of a request. We base our estimation on the real operation of an ISN server in a search engine. A working thread on an ISN server has to score the documents of a query's posting list [32] one by one to retrieve the top-K relevant results [36, 11]. This provides an opportunity to precisely estimate a request's service time. Although some documents on a query's posting list might be skipped due to pruning techniques [38, 9], an advanced prediction model with multiple features can be developed to improve accuracy [72]. We propose a NN-based model that incurs only 0.79% additional delay on our platform, which is less than the centralized control overhead in Pegasus [69] or similar to the control theoretic circuit in [48]. Query specific latency prediction has been proven useful to significantly reduce the 99.99th percentile tail latency on Microsoft Bing [61]. Even then, this prediction is likely to have some errors. Therefore, in Gemini, we use a second NN model to predict the error. The sum of predicted latency and error estimate for a query is used to determine

the correct frequency boosting time for the two-step DVFS scheme.

For medium and high server loads with request queuing, we boost the current CPU frequency whenever a critical request arrives at the queue. A critical request is the one for which the target latency is violated if we adhere to the current DVFS setting. In the general case with more than one request in the queue, they all will use two-step DVFS until a critical request arrives. All the other requests in between execute at the current frequency, to minimize the frequency transition overheads. Requests using the same frequency have already met their latency constraints before the critical request arrives, and we observe that only 1.65% of requests have wrong estimations for latency plus error. The arrival of critical request results in a higher current frequency. Thus, requests in-between will finish their tasks before the deadline.

To validate our design, Gemini is implemented in the commercial Solr search engine [101], which is deployed on a 24-core platform with user-space per-core DVFS control and power management. Experimental results on three different query traces prove that Gemini saves 41% of CPU power, which is 1.53 times better than Rubik [60] and 2.05 times better than Pegasus [69]. At the same time, Gemini produces the minimum deadline violation rate compared to Pegasus and Rubik. Our major contributions are the following:

- A novel heuristic two-step DVFS is proposed, which properly boosts the CPU frequency to catch-up to deadlines, on a per-query basis. The frequency steps are adjusted at the arrival and departure of a critical request when there are multiple requests in the queue.

- A low overhead NN model is developed to precisely predict the service time at a

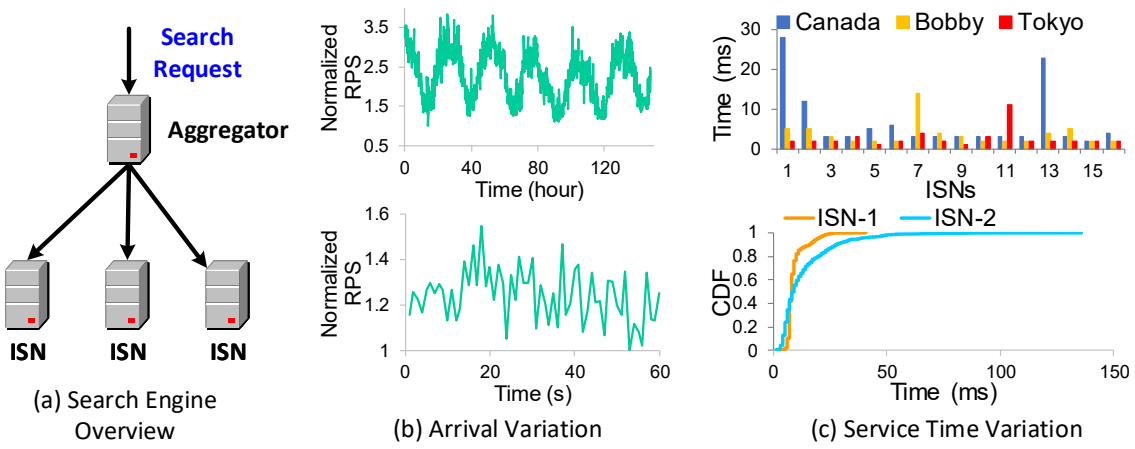


Figure 6.1: Search workload exhibits high variations. In part (b), the top figure presents the diurnal and day-of-week pattern for the normalized RPS; the bottom figure shows the arrival variations in a short term.

per-query granularity based on realistic features in query processing. Additionally, a separate error prediction model is utilized to determine the exact time for boosting the frequency in our two-step DVFS design.

- Finally, Gemini is implemented on a real platform using the commonly used Solr search engine in production. Three representative query traces are evaluated for energy saving and compared with previous techniques.

6.2 Search Workload Characteristics

As shown in Fig. 6.1 (a), a search engine typically employs a partition-aggregate architecture. Data collections in a search engine often contains billions of documents [56, 32], which are partitioned into a number of shards. Typically, one shard of documents is served by an ISN server. The ISN organizes its local documents as an inverted index [132] (similar to hash-map), in which each key in the query term dictionary is linked with a list of matched

documents (i.e., the query term's posting list). Whenever the aggregator receives a search request from a client, it will broadcast the request to all the ISNs. Every ISN then searches its local index performing a scoring of documents in the posting list. As there are millions of documents in an index shard, only the top-K scored documents are sent back to the aggregator. Finally, the aggregator merges and ranks all the responses from the ISNs before sending the search results to the client. The overall latency for a search result is limited by the slowest response arriving from the ISNs. Thus, it is critical to meet the strict tail latency constraint at the ISN servers, for good search latency and quality. The stragglers will by ignored by the aggregator of the responses.

Fig. 6.1 (b) shows that the incoming request rate to a search engine can exhibit high variation over both long and short time intervals. The top sub-figure plots the RPS rate for a Wikipedia query trace [107] over a period of 150 hours. All the data points are normalized to the smallest observed RPS. The RPS of search queries follows a diurnal and day-of-week pattern. For power management, a coarse granularity epoch based latency prediction can capture this workload variability [69]. However, results on the bottom sub-figure shows that the Wikipedia query trace also has a high per second granularity RPS variability, in addition to the longer term variability. This suggests that it is necessary to have a per query basis design for the search engine power management.

To account for the variation of incoming requests, prior works [60, 23] adopt an analytical model, which considers the queuing, for power management, upon every request arrival and departure. To estimate the per request *equivalent latency* (i.e., service time plus queuing time), they assume that each request's service time can be derived from the

same distribution. Nevertheless, the results in Fig. 6.1 (c) invalidate this assumption. The top sub-figure presents the measured request service times on a search engine with 16-ISNs which is deployed on our testbed platform. We find that the service times for three consecutive queries (i.e., Canada, Bobby and Tokyo) vary a lot on the same ISN server. For example, the service time of query "Cananda" is 14 times longer than query "Tokyo" on ISN-1. The CDF of service time in the bottom figure confirms that this kind of variability exists across a wide range of 20K requests we measured. Although their analytical model can capture the arrival variation, it can not fully utilize the per query service time variation. Our experimental results indicate that each query's service time depends on the particular query's features and the posting list at the particular ISN, which itself can vary quite widely.

6.3 Two-Step DVFS

In our power management scheme, some requests will leverage a two-step DVFS method to catch-up to the deadline through boosting at the second step. To make the analysis easy to understand, we first describe our design with the assumption that there is always one request in the queue. Then, we extend it to two requests, and finally to the general case of N requests.

6.3.1 Single Request Without Queuing

Fig. 6.2 shows a two-step DVFS control for a single request. In Fig. 6.2 (a), request R_1 arrives at time A_1 and has to finish its work before its deadline D_1. Fig. 6.2 (b) shows that the frequency is boosted to the maximum frequency at time T_1 in order to catch

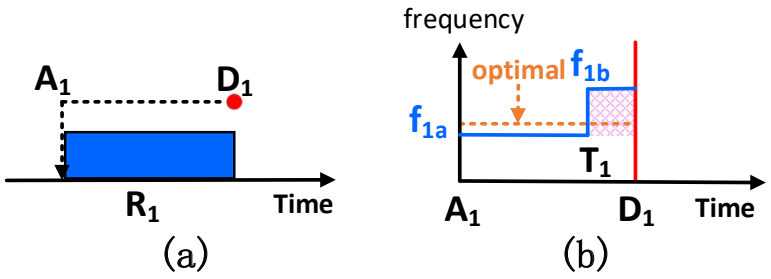


Figure 6.2: Example of Two-Step DVFS Control for single request in the queue.

up the query execution so as to finish at D_1. Suppose the CPU is running at frequency $f_{default}$ before R_1 enters the queue. For power management, we predict request R_1's service time with our NN model. The predicted service time S_1^* is:

$$S_1^* = Predict^{NN}(Q_1, I | f_{default}) \tag{6.1}$$

where Q_1 is the request's query and I is an ISN's index. Given Q_1 and I, we can obtain request R_1's query features. All the predicted service times are conditioned by the default CPU frequency $f_{default}$. Previous works [60, 23, 69] assume that a search request's service time S is inversely proportional to the CPU frequency. Specifically, Chou et al. in [23] reports that a request's total work $W = Mf + C$, where C is the CPU cycles and M is the *memory access time*. Therefore, $W/f = M + C/f$ where f is frequency. Since we focus on the CPU management, we omit the memory access time M, treating it as a constant, as in prior work [23] and [60]. For simplicity, we use the service time S to refer to C/f. Then, the aforementioned equation becomes $S = C/f$. To validate this equation, we measured a particular search query's latency at various CPU frequencies on our experimental platform. In Fig. 6.3, we see that the request's latency increases from 40ms to 97ms when the CPU frequency is slowed down from the fastest 2.7GHz to the

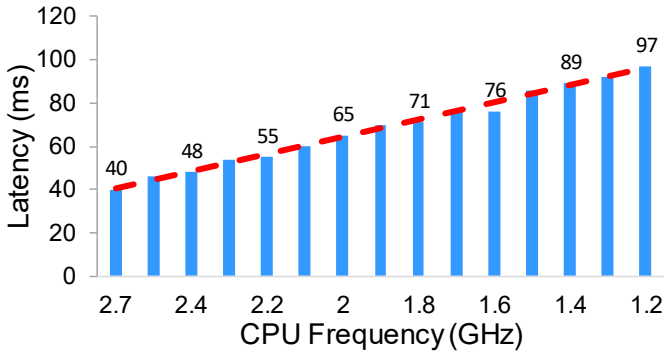


Figure 6.3: Search query processing is CPU-intensive: service time is inversely proportional to CPU's frequency.

slowest 1.2GHz frequency. Additionally, we fit a latency trending line on the same figure. Almost all the $< frequency, latency >$ sample points are exactly on this linear trending line, confirming that the compute-intensive search query processing time has a linear dependence to CPU frequency. The time delay for the CPU to transition from one frequency to another with the CPU stalls, is a constant, T_{dvfs}. T_{dvfs} is typically around 10 microseconds in the kernel and 30 microseconds in user-space [4]. This time is relatively small compared to the tens of milliseconds for the service time. If the predicted service time S_1^* equals R_1's actual service time S_1 with 100% prediction accuracy, the frequency set during the time interval A_1 to D_1 should be constant [70] (i.e., the dotted line in Fig. 6.2 (b)). This optimal frequency f_1 can be calculated as follows:

$$W_1 = W_1^* = S_1^* * f_{default} \tag{6.2}$$

$$f_1 = W_1^*/(D_1 - A_1 - T_{dvfs}) \tag{6.3}$$

where W_1 is request R_1's total work and W_1^* is the predicted value. However, a prediction is likely to not be 100% accurate. Accounting for this, we have the following:

$$S_1^* = S_1 + E_1 \tag{6.4}$$

where E_1 is the prediction error. With unknown S_1 during runtime, a step-wise DVFS control can produce better power savings [70, 118]. To minimize the DVFS transition overhead, we limit our scheme to be a two-step design. For a two-step DVFS, shown in Fig. 6.2 (b), we have to solve three problems: 1.) select the initial frequency f_{1a} 2.) determine the time T_1 for frequency boosting and 3.) choose the boosted frequency f_{1b}. In Gemini, we fix the boosted frequency at maximum for the CPU core and use the predicted service time S_1^* to select the f_{1a}. Then, f_{1a} is:

$$f_{1a} = S_1^* * f_{default}/(D_1 - A_1) \tag{6.5}$$

In the following analysis, we focus on the case when S_1^* is shorter than S_1, as a deadline violation is more serious than energy inefficiency. If the prediction of S_1^* is accurate, the line of f_{1a} in Fig. 6.2 (b) would be straight (to D_1). Nevertheless, the shaded area reflects period we have to boost to f_{1b} to accommodate for prediction errors so that we meet the latency requirement.

To find the correct value of T_1, we choose f_{1b} to be equal to $f_{default}$ since we want the CPU frequency to stay at the lower f_{1a} for as long as possible. Since we do not know S_1 precisely during the runtime, we design a separate error predictor to help us in this process. The output of our error predictor E_1^* for request R_1 and predicted service time S_1^* are used to approximate the actual service time S_1. E_1^* can be obtained by the following equation:

$$E_1^* = Predict^{Error}(Q_1, I|f_{default}) \tag{6.6}$$

Intuitively speaking, we leave a little latency slack for prediction errors. Then, the following equation holds.

$$f_{1a} * (T_1 - A_1) + f_{1b} * (D_1 - T_1 - T_{dvfs}) = (S_1^* + E_1^*) * f_{default} \tag{6.7}$$

Notice that we execute at frequency f_{1b} for a time interval $D_1 - T_1 - T_{dvfs}$, since the CPU will stall for T_{dvfs} whenever we change the frequency. By combining equations 6.5 and 6.7, T_1 can be calculated. Generally, there is a trade-off between power and deadline violations. However, our violation rate can be fixed at a much lower value because of the accurate error predictor. In the worst case, T_1 will be at the beginning, A_1, where we have to boost the frequency right away to meet the deadline. If T_1 is equal to A_1 and request R_1 still cannot finish its work W_1 before the deadline D_1, our DVFS scheme will directly drop request R_1 to save energy. Dropping this kind of request will not impact the search quality seen by the client [123], as this response would be dropped by the aggregator anyway [123, 16]. The accuracy of the service time prediction is very important to the power saving achieved with our two-step DVFS. In equations 6.5 and 6.7, f_{1a} will be closer to optimal frequency f_1, and boosting time T_1 will be closer to deadline D_1, if the service time prediction error is smaller. When the prediction error E_1^* is zero, we have $f_{1a} == f_{1b}$ (i.e., use a constant frequency).

6.3.2 Two Requests With Queuing

The scenario when two requests might stay in the queue is more complicated as we need to re-configure current CPU frequency to guarantee the request R_2's latency requirement. Fig. 6.4 (*Case 1a*) depicts the arrival of the second request R_2 at time A_2

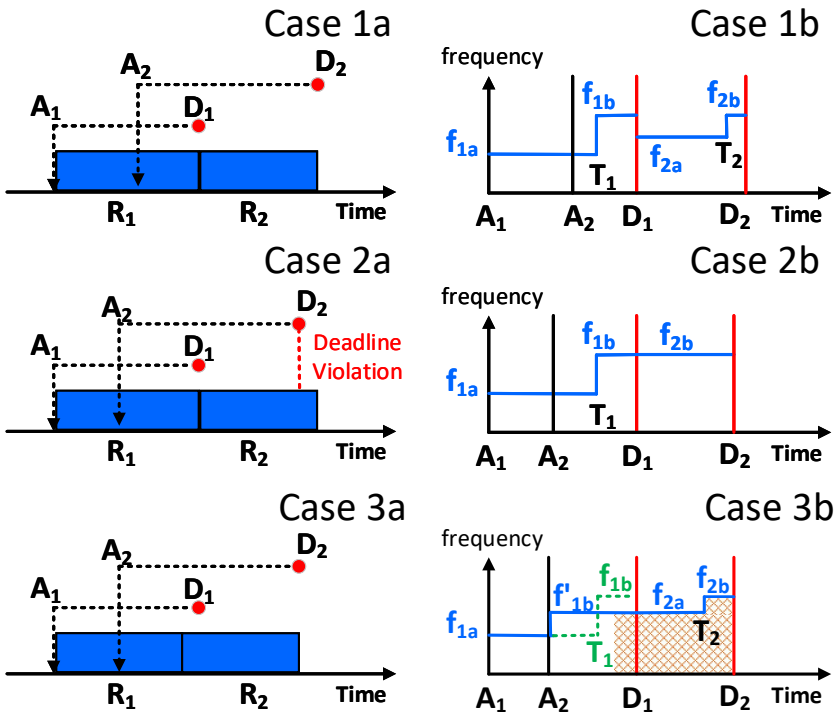


Figure 6.4: Two requests in the queue. Example of Critical Request and Non-Critical Request.

after R_1 started processing at time A_1. The deadlines for the two requests are D_1 and D_2, respectively. When the inter-arrival time, $A_2 - A_1$, is large enough as in Fig. 6.4 (*Case 1b*), such that request R_2 can finish its job within the interval D_1 to D_2, we call request R_2 non-critical. When a non-critical request arrives at the queue, we don't need to re-configure the current setup and request R_2 still uses a two-step DVFS to save power. On the other hand, request R_2 in Fig. 6.4 (*Case 2a*) will violate its deadline when the interval between D_1 to D_2 is too short. In *Case 2b*, request R_2 is considered to be a critical request when the following happens:

$$(D_2 - D_1) * f_{2b} < (S_2^* + E_2^*) * f_{default} \tag{6.8}$$

$W_2^{'} = (S_2^* + E_2^*) * f_{default}$ is the total amount of work predicted for request R_2, including prediction errors. When request R_1 finishes, we skip step one for request R_2 and directly boost the CPU frequency to f_{2b} (i.e., $f_{default}$). With that being the case, the maximum amount of work that can be done within the residual time $D_2 - D_1$ is $(D_2 - D_1) * f_{2b}$. Request R_2 is critical when $(D_2 - D_1) * f_{2b}$ is smaller than $W_2^{'}$. In order to guarantee R_2's latency constraint in Fig. 6.4 (*Case 3b*), we have to boost the current frequency immediately and re-configure the second step frequency f_{1b} (green line) to $f_{1b}^{'}$ (blue line), in order to finish R_1 early. Then, request R_2 (i.e., orange shaded area in *Case 3b*) can begin to execute even before D_1. Notice that we can boost the frequency earlier only when the arrival time A_2 of request R_2 is earlier than the initial boosting time T_1. Otherwise, nothing can be adjusted as frequency f_{1b} in *Case 3b* is already $f_{default}$, when A_2 is between the initial value of T_1 and D_1. To minimize the transition overhead, we select the same frequency for $f_{1b}^{'}$ and f_{2a}. If a combination of $< f_{2a}, T_2, f_{2b} >$ can make request R_2 meet its deadline D_2, then request R_1 will definitely complete before the deadline D_1. So, we now focus on the calculation of $< f_{2a}, T_2, f_{2b} >$ for request R_2. Before selecting the frequencies and boosting time for R_2, we define its predicted equivalent CPU cycles eW_2^* as follows:

$$eW_2^* = (S_1^* + E_1^*) * f_{default} - (A_2 - A_1) * f_{1a} + S_2^* * f_{default} \tag{6.9}$$

where $(S_1^* + E_1^*) * f_{default} - (A_2 - A_1) * f_{1a}$ is request R_1's residual work and the $S_2^* * f_{default}$ is request R_2's predicted work. With equivalent total work eW_2^*, the initial frequency f_{2a} for request R_2 becomes:

$$f_{2a} = eW_2^* / (D_2 - A_2 - T_{dvfs}) \tag{6.10}$$

Similar to the single request design, we let f_{2b} be $f_{default}$. In order to obtain the boosting time T_2, we have the following equation:

$$\begin{aligned} f_{2a} * (T_2 - A_2 - T_{dvfs}) + f_{2b} * (D_2 - T_1 - T_{dvfs}) \\ = eW_2^* + E_2^* * f_{default} \end{aligned} \tag{6.11}$$

where we must finish the total work for the two requests before deadline D_2. By combining equation 6.10 and 6.11, the boosting time T_2 can be calculated. A special scenario of *Case 3b* is when an incoming request R_2 cannot finish its work even if we boost the CPU frequency to $f_{default}$ immediately after it arrives. In such a case, it is safe to just directly drop request R_2, in the interest of saving more energy.

6.3.3 General Case with N Requests

We now address the general case. There are $N-1$ requests in the queue when critical request R_N arrives. If request R_N is non-critical, no action needs to be taken. In Fig. 6.5 (*Case 1a*), we give an example with $N = 3$. Request R_1 currently uses a two-step DVFS to save power and the next request R_2 is non-critical. When the critical request R_3 arrives at the queue in *Case 1a*, we have to boost request R_1's initial frequency f_{1a} to f'_{1b} as in Fig. 6.5 (*Case 1b*). Similar to equation 6.9, request R_3's equivalent total work eW_3^* considering R_2's work has to be calculated before making the new frequency plan. In general, request R_N's equivalent total work eW_N^* is:

$$eW_N^* = W_1^{residual} + \sum_{1<i<N} (S_i^* + E_i^*) * f_{default} + S_N^* * f_{default} \tag{6.12}$$

where request R_1's residual work $W_1^{residual}$ means:

$$W_1^{residual} = (S_1^* + E_1^*) * f_{default} - (A_N - A_1) * f_{1a} \tag{6.13}$$

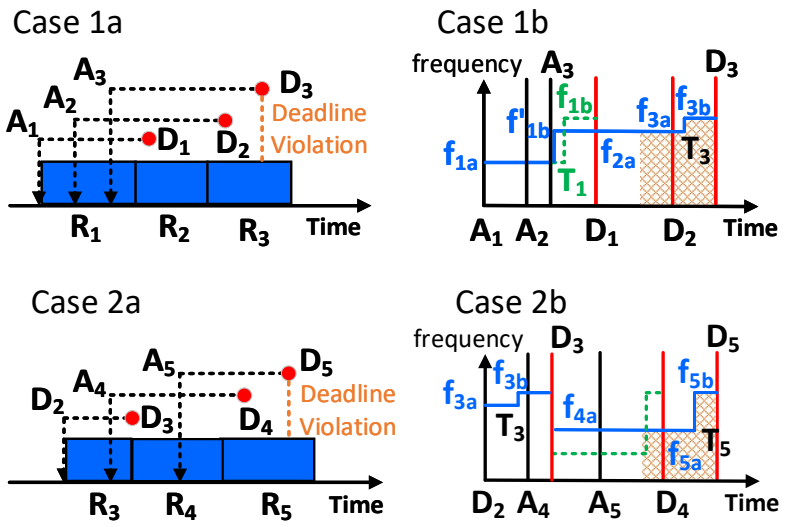


Figure 6.5: Example of N (=3) requests in the queue. In Case 2a, request R_3 and R_4 can finish before their deadlines if we use f_{3a} as the current CPU frequency. After R_3 leaves, R_5 suffers the deadline violation if R_4 adopts a two-step DVFS without considering the later request R_5.

The new frequency f'_{1b}=f_{2a}=...=f_{Na} can be calculated using the same method described in the case of Two Requests. Thus, we have the following equation:

$$f'_{1b} = f_{Na} = eW^*_N/(D_N - A_N - T_{dvfs}) \tag{6.14}$$

In this design, all the requests in between current request R_1 and critical request R_N adopt the same frequency to minimize the frequency transition overhead. Before the critical request R_N arrives, the current frequency plan has already guaranteed that the existing $N-1$ requests in the queue can meet their deadlines. Due to the arrival of critical request, we have to boost the current frequency. So, request R_1 to R_{N-1} will finish before their deadlines. It is safe to use the same frequency for them to reduce transition overhead. After determining f_{Na} from equation 6.14, we use the following equation to get T_N, with

f_{Nb} to be $f_{default}$.

$$f_{Na} * (T_N - A_N - T_{dvfs}) + f_{Nb} * (D_N - T_N - T_{dvfs}) = eW_N^* + E_N^* * f_{default} \tag{6.15}$$

Fig. 6.5 (*Case 2a*) shows the scenario when request R_1 and R_2 depart from the queue. Currently, request R_3 is still critical and new arrivals of request R_4 and R_5 are non-critical. In Fig. 6.5 (*Case 2b*), the non-critical request R_4 might adopt a two-step DVFS (green line) to save its power. However, request R_5 in *Case 2a* suffers a deadline violation if R_4 only considers its own computation demand. In Gemini with N requests in the queue, we plan the CPU frequency in groups. For example, we find the next critical request (R_5) in *Case 2a* after the current critical request R_3 departures. Then, our design uses the method in *Case 1* for selecting the frequencies such that we can meet the deadlines and reduce transition overheads.

6.4 Latency and Error Predictors

For power managements, inaccurate service time estimation can result in deadline violations, when we attempt to finish the requests just-in-time [60, 23]. Gemini employs two NNs to estimate the query specific service time and prediction error, respectively. As described in last section, the predicted service time S^* is used in the initial frequency selection in our two-step DVFS, and the error predictor is for determining the boosting time T.

Table 6.1: Features for the Service Time Prediction

Query	**Time (ms)**	AMean Score	GMean Score	HMean Score	Max Score	Estimated MaxScore	Score Variance
Toyota	13	9.34	9.05	8.68	14.81	1131	5.99
Federal	29	7.11	7.1	7.1	7.9	1311	0.05
United Kingdom (*Max*)	21	6.9	6.9	6.89	7.42	2144	0.02

Query	**Time (ms)**	# of Postings	# of Local Maxima	Local Maxima above AMean	# of MaxScore
Toyota	13	20742	3084	2639	1
Federal	29	497081	3135	2506	1
United Kingdom (*Max*)	21	2369024	2834	2373	1

Query	**Time (ms)**	Docs in 5% Max Score	IDF	Docs in 5% K^{th} Score	Docs ever in Top-K	Query Length
Toyota	13	199	6.81	322	85	1
Federal	29	608	3.64	805	72	1
United Kingdom (*Max*)	21	6357	3.41	9946	81	2

6.4.1 Latency Prediction

Recent research [55, 61, 72] in the *Information Retrieval* area reports that with the adoption of Selective Pruning [15, 36] in search engines such as Facebook Unicorn [32] and Microsoft Bing [55], a simple Linear Model [80] is inadequate to get a precise per query service time prediction. Linear Classifier utilizes a linear combination of features to make predictions, and therefore has poor accuracy. A few researchers have developed sophisticated machine learning models to predict the query latency [55, 61, 72]. We consider many features in query processing and limit our selection to a few important ones. Table 6.1 shows, across the different columns, all the features used in our prediction model. We have chosen to illustrate these features with example term queries (Toyota, Federal) and phrase queries (e.g., United Kingdom). The service time in column 2 is the prediction label

in our model. All the other columns are prediction features. AMean score is a query term's arithmetic average score of all documents in the posting list. Similarly, the GMean score is the geometric mean score and HMean is harmonic mean. Estimated MaxScore is an approximation of the max score based on the algorithm in [71]. In a query term's score distribution, there might exist local maxima. The # of Local Maxima, and Local Maxima above AMean, capture these score distribution features. Additionally, we quantify the number of documents that fall within 5% of max score and 5% of the K^{th} score, where K is the size of result sets. Finally, the feature "Docs ever in Top-K" is the number of documents that are fully scored by the selective pruning algorithm. For those phrase queries having more than one term, the maximum of query terms' feature values is used.

In Gemini, we use a NN model *with only 5 hidden layers* to make the latency prediction, because it achieves a good balance between prediction accuracy and inference overheads, on our platform. Each hidden layer has 128 neurons and uses the relu activation function. Our classification model is trained by the Adam optimization algorithm [63] with sparse categorical cross-entropy loss function. For service time scaling in Gemini DVFS, our NN model predicts each query's service time using the default CPU frequency. Next, we evaluate the feature importance on NN model. In this experiment, we first develop a NN model with #_of_Postings as the only feature. Then, more features are added to this NN model in the order of top to bottom as shown by the Y-axis of Fig. 6.6. All the features listed on the Y-axis of Fig. 6.6 are from Table 6.1. In Fig. 6.6, we can observe that the NN model with single feature (i.e., #_of_Postings) only yields a prediction accuracy of 23%. When having more features, the prediction accuracy improves. Finally, the model with all

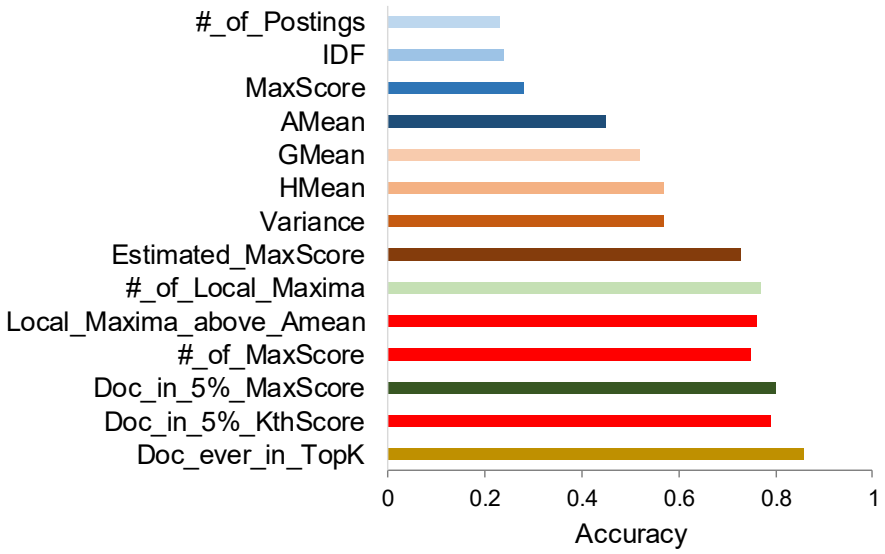


Figure 6.6: The prediction accuracy when we keep adding new features. Red bars are query features that will adversely impact the prediction accuracy.

the query features at the bottom of Fig. 6.6 achieves 89% prediction accuracy. Red bars in the figure, such as Local_Maxima_above_Amean, #_of_MaxScore and Doc_in_5%_KthScore, are query features that will degrade the prediction accuracy. In Gemini, we carefully select the features for a search engine system such that our NN model achieves the maximum prediction accuracy.

6.4.2 Model Comparison

We now compare the prediction error and overhead results for the NN classifier, NN regressor and a simple linear classifier on the Wikipedia query trace and index. All the models are using the same query features. During the experiment, we observe that the prediction error will reduce if we train the models over more iterations. The detailed comparison results are given in Fig. 6.7. In Fig. 6.7 (a), each model's X-axis value is its

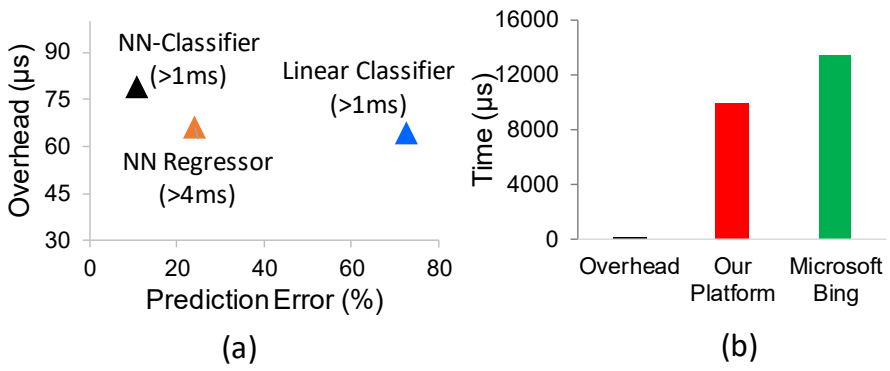


Figure 6.7: (a) Prediction error and overhead for the NN models and the linear model. (b) Prediction overhead and average request service time.

prediction error and the Y-axis value is its prediction overhead. For the NN regressor with the MSE loss function, we train it by using the RMSprop optimization algorithm. Due to the selection of a MSE loss function, we define that a prediction error happens when the absolute value of true service time minus the predicted service time is larger than 4ms. On the other hand, the threshold for the (NN and linear) classifier models is 1ms as the output neuron of our classification model is at per millisecond granularity. Although the simple linear classifier has the smallest overhead of 64 μs, its worst prediction error of 73% prevent us to use it. Next, the NN regressor has a better prediction error of 24% with similar prediction overhead (66 μs). However, the 24% prediction error is still too high for the search request processing with strict latency constraints. Finally, the selected NN-classifier in Gemini has the smallest prediction error of only 11%. The 79 μs of prediction overhead in our NN classifier is relatively small compared with a search request's total service time. In Fig. 6.7 (b), we plot the average request service time on both our platform and Microsoft Bing [55]. On our platform, the average request service time is around 10000 μs (i.e., 10ms). Microsoft reports that the average service time for Bing search is 13470 μs

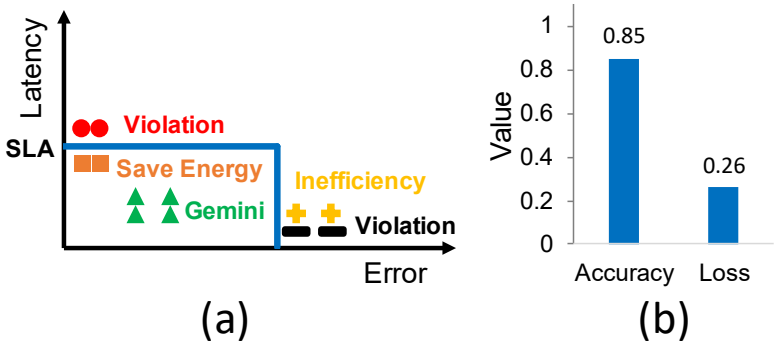


Figure 6.8: (a) The impact of latency and error prediction for energy management. (b) accuracy and loss of our error predictor.

(i.e., 13.47ms) due to the large index [61, 17]. The average request service time is 127 times of the prediction overhead for our NN classifier.

6.4.3 Design of an Error Predictor

Finally, we investigate the prediction error E of our NN model. In Gemini, the prediction error E is defined as $f(X_i) - y_i$, where we derive the predicted service time as $f(X_i)$, given the feature vector X_i and y_i is the measured query service time, for query i. Prediction results show that 89% of requests have accurate service time prediction. However, around 5.5% of requests have positive prediction errors and the remaining 5.5% requests see negative prediction errors.

The impact of this kind of prediction errors is discussed in Fig. 6.8 (a). The X-axis on Fig. 6.8 (a) is the prediction error and the Y-axis is a request's predicted latency. If the prediction error is negligible, a long request (red circle) may violate the SLA requirement while a short request (orange square) may be slowed down for power saving and still not violate the SLA. However, even for such a short request, if the prediction error is negative

and large (shown with black dashes), this will result in the predicted service time $f(X_i)$ being much shorter than y_i, causing us to select a much lower CPU frequency than appropriate and result in a violation of the SLA. Similarly, large positive prediction errors (yellow plus sign) will result in energy inefficiency because we will select too high a frequency. In order to solve this problem, Gemini uses an additional predictor to estimate a request's service time prediction error [61]. For the predicting the error, we use the same query features as listed in Table 6.1. To train another NN model for error prediction, the label $E = f(X_i) - y_i$ can be easily obtained in training set since we can keep track of the measured request latencies in the past. Fig. 6.8 (b) shows the prediction accuracy for errors is 85%.

6.5 Gemini Implementation

Gemini is implemented on the Solr search engine. The high level architecture of Solr search engine is shown in Fig. 6.9. When a client's search request arrives, a SearchHandler inside the Solr search engine will forward the request to the IndexSearcher component, which calls the Apache Lucene APIs for retrieving the relevant documents. The entire Solr search engine is written in Java. To implement Gemini, we wrap the IndexSearcher and Lucene Index Searching in Fig. 6.9 as a Java Callable task. The reason for doing this is that the Java Executor framework can automatically handle Callable tasks in a Blocking Queue and provide mechanisms for thread management. When a search request arrives, we submit its task to the Blocking Queue and wait for an idle working thread to process its query. Currently, our implementation only has one working thread as we focus on single core power management. With multiple CPU cores, we can maintain a separate queue for

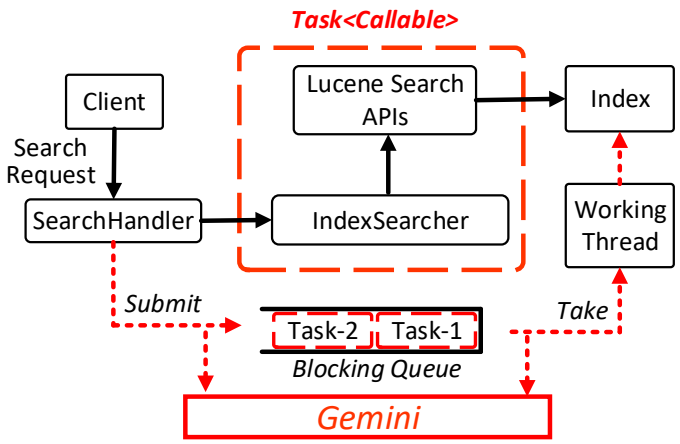


Figure 6.9: Gemini implementation overview.

each core and have a global broker to distribute the incoming requests to each core, as suggested in [72]. Each core will manage its power consumption independently by using Gemini's DVFS scheme. This is work we plan for the near future. Finally, we conduct the ISN power management upon the arrival and departure of every task.

We use the TensorFlow [1] to achieve Gemini's NN prediction model. The inference time for prediction is only 73 - 83 microseconds, compared with tens of milliseconds for the query's service time. As Gemini is implemented as a part of the ISN, we need user-space DVFS control. Gemini leverages the Advanced Configuration and Power Interface (ACPI) to update the CPU core's frequency at runtime. To reduce transition overheads, our frequency enforcement is achieved by manipulating each core's "scaling_setspeed" file. When this device file is changed, Linux triggers a group of system calls that take only 40 microseconds totally to update the CPU core's frequency. The extra latency overheads from the Gemini implementation are negligible.

Our experimental setup has two machines connected by a 1G Ethernet link: one

as the client and the other as the search engine server. The server machine is a 24 core Intel Xeon E5-2697 CPU, 128G memory running CentOS 7 operating system. Three representative query traces are used in our experiments: the Wikipedia [107], the Lucene nightly benchmark [48] and the TREC Million Query Track (MQT) [81]. On the server side, we deploy a 12 ISNs Solr search engine. In the search engine, we index the complete dump of entire English Wikipedia web pages on December 1st, 2018. This 65GB index has a total of 34 millions documents. The search engine machine supports per core frequency scaling. The CPU frequency can be selected in the range of 1.2 GHz to 2.7 GHz. The default CPU frequency is 2.7 GHz. For power measurement, our CPU has sensors to monitor per socket's energy consumption. The accumulated energy consumption of a CPU socket is stored in a Machine Specific Register (MSR). By writing a energy measurement daemon which reads the MSR register every 1 second through the Running Average Power Limit (RAPL) interface, we can obtain a CPU socket's power consumption. As the measured CPU energy per second is the socket package energy, we deploy 12 single-working-thread ISNs on the 12-core CPU chip. Then, each ISN is bound to one core of the CPU chip. The 12 ISNs receive the same search queries from our Solr aggregator but schedule their core's frequency independently.

6.6 Evaluation

Gemini is first evaluated with a range of server loads, in terms of RPS, to show the significant CPU power savings while still achieving the latency constraints. We also perform trace-driven experiments using three different query traces to characterize Gemini's power

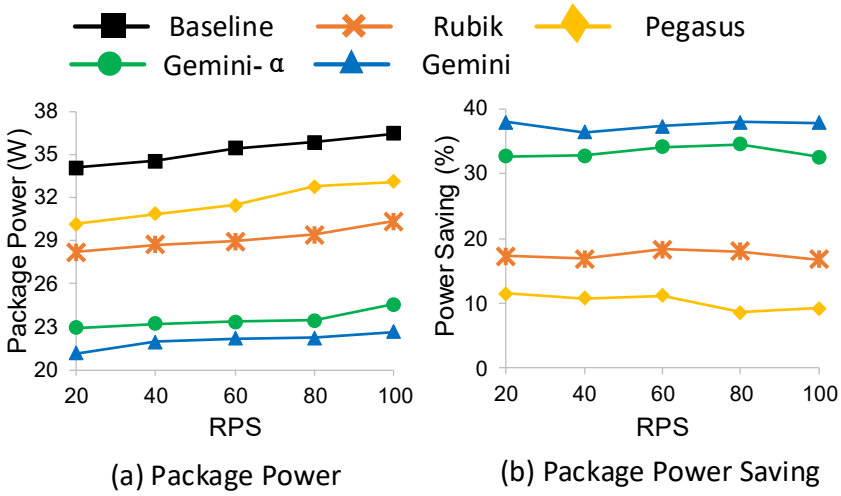


Figure 6.10: The CPU package power results for various RPS. Time budget uses the tail latency at high load.

management under realistic workloads. Finally, we disable each component of Gemini in turn, to show the underlying reasons for the increased power savings of Gemini.

6.6.1 Power Saving

In this experiment, we implement four different frameworks for comparison. As shown in Fig. 6.10, the baseline doesn't have any power management and always uses the default 2.7 GHz CPU frequency. The first is Rubik [60], an analytical power management framework which adjusts the CPU frequency on every request arrival and departure according to the tail (95th percentile) of the service time distribution. Second, we compare with Pegasus [69], which is a feedback based power management scheme. It measures the request's latency periodically and selects the highest CPU frequency if a deadline violation happens. To match the search load we use over 1000 seconds, we scale Pegasus's 5 seconds epoch length (over a 12-hour period for the load) to 125 milliseconds in our experiment, so

as to have the same ratio between epoch length and load length. Apart from the complete design of Gemini, we also implement another version of our scheme called Gemini-α, in which we don't use the second NN model for error prediction. Instead, we use the moving average of errors in the past 60 request arrivals to estimate the current request's latency prediction error. By doing this, we quantify the benefit of having the second predictor for the error in our design on CPU power saving and request tail latency.

To compare across the different power management frameworks, we use the same query workload (the Wikipedia trace) on the Solr search engine. The results are shown in Fig. 6.10. Each request rate (in RPS) is maintained for 120 seconds to obtain the corresponding average CPU power consumed at that server load. Fig. 6.10 (a) shows the CPU package powers for RPS varying from 20 to 100. In our experiment, the measured CPU package power includes the power consumed by CPU cores, caches and other components on the chip. The results show that CPU package power increases for all the frameworks, as we increase the request intensity. For example, the CPU package power for the baseline case increases from 34W to 36.5W when the RPS value goes up from 20 to 100. Higher server load results in more request queuing. Thus, the latency slack that power management frameworks used to slow down requests is reduced. Among all the power management alternatives, Pegasus consumes the highest CPU package power, because of its epoch based design. Rubik performs better than Pegasus, but still consumes more CPU package power than our Gemini variants. The results in Fig. 6.10 (a) shows that the complete design of Gemini uses the least CPU package power and outperforms the Gemini-α which doesn't include a second error predictor. The reason is that the inaccurate moving average prediction

in Gemini-α makes our two-step DVFS enter the second frequency step too early in its goal of meeting the latency constraint. So, Gemini-α consumes more CPU power than Gemini across the entire range of loads. The second error predictor in Gemini is clearly beneficial in reducing CPU power consumption.

Fig. 6.10 (b) presents the package power saving in percentage compared to the baseline. Although the actual power consumption increases linearly in 6.10 (a) for each technique, the percentage power savings remains same for each technique compared to the baseline. As the relative differences across frameworks are similar for the range of RPS, we focus on the high server load of 100 RPS, where power savings are more challenging to achieve. At this high server load, Pegasus saves 9.23% of CPU package power while Rubik achieves 16.8% power saving compared to the baseline. The power savings for Gemini-α is 32.7%, which is 1.95 times better than Rubik. The complete design of Gemini performs the best, with 37.9% CPU package power saving compared to the baseline. This is 2.25 times better than Rubik, and is even better when compared to Pegasus.

6.6.2 Tail Latency

For the same experiment as above, the 95th percentile latency for different RPSs is shown in Fig. 6.11. In Fig. 6.11 (a), we observe that the tail latency for the baseline, with a fixed 2.7 GHz frequency, increases with load, due to request queuing. For example, the request's tail latency is 12.3 ms at 100 RPS, compared with the 7.05 ms at 20 RPS. To utilize the latency slack, and save CPU power, power management frameworks such as Rubik and our Gemini will slow down the request's processing time to be close to the

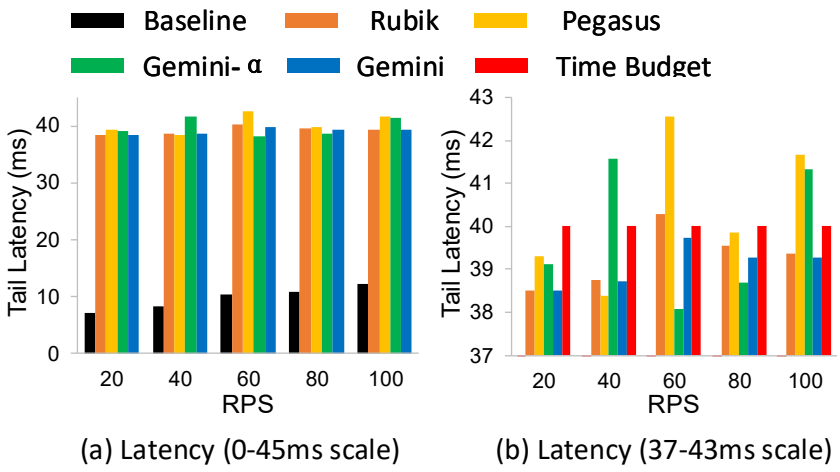


Figure 6.11: Tail latency results for different RPS. Part (b) is the tail latency with 37-43ms scale for part (a).

40 ms latency bound. Fig. 6.11 (a) shows that all the compared frameworks can roughly achieve this performance constraint. We look into the tail latency at a finer granularity (in the 37-43ms range) under the different frameworks, Fig. 6.11 (b), with the red bar in the figure being the time budget given. At 100 RPS, the tail latency of Rubik is 39.36ms and the others are similar, across the range of RPS. Because Pegasus has a feedback based controller to select CPU frequency, it has a higher tail latency variations across the range of loads. Pegasus violates the 95th tail latency constraint by achieving 41.67ms at 100 RPS. Due to the inaccurate error estimation in Gemini-α, which uses a simple moving average from recently processed queries, Gemini-α sometime has a higher 95th tail latency than the target time budget of 40 ms. Finally, the complete Gemini along with the error predictor consistently meets the 40ms tail latency constraint all the time. Compared with Rubik and Pegasus, Gemini meets the deadline requirement but still achieves higher CPU power savings, as shown in Fig. 6.10. This is because we accurately "reshape" the request's latency distribution, so that more of the requests have their latency closer to the time budget.

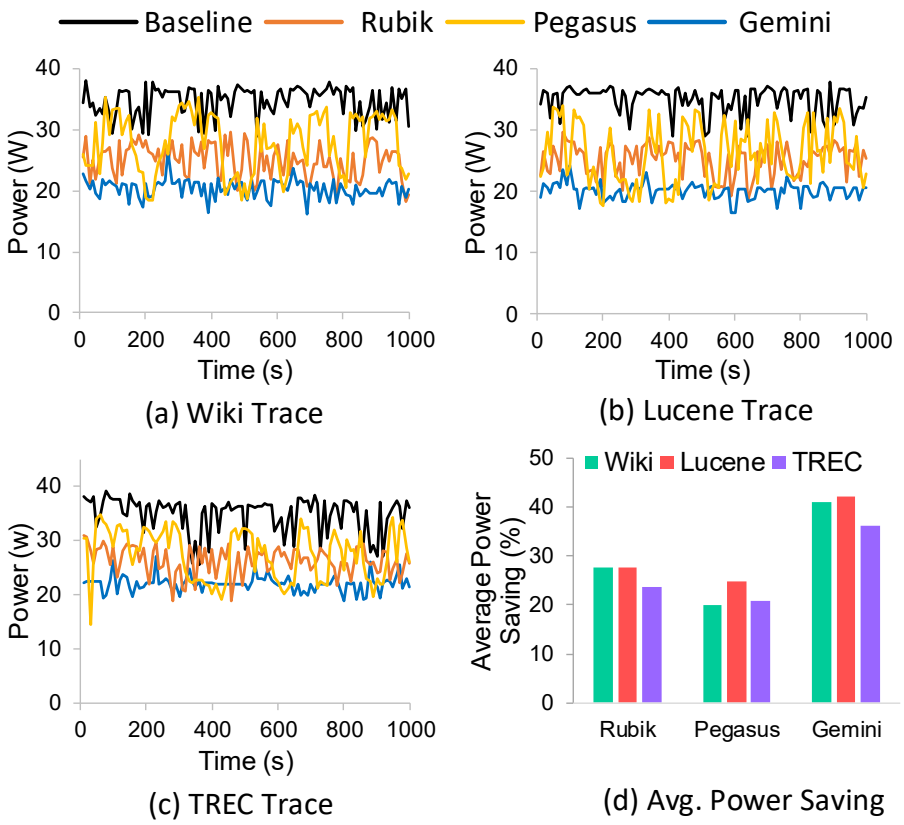


Figure 6.12: Power consumption results for the Wikipedia, Lucene and TREC traces.

6.6.3 Trace-Driven Characterization

For the next experiment, we evaluate Gemini with a constant RPS trace-driven workload, for sensitivity analysis. We compare Gemini with Rubik and Pegasus through a 1000 seconds trace-driven experiment. Fig. 6.12 (a) and (b) present the CPU package power for the Wikipedia and Lucene nightly benchmark traces, respectively. Similar to prior works [81, 33, 65], we also show the comparison results for the TREC query traces (widely used in the information retrieval area), in Fig. 6.12 (c). For these three query traces, we still use 40ms as the target tail latency. With all the traces, power consumption of the baseline is in the range of 29.1W to 38.2W as the server load varies over time. Pegasus still consumes the

most CPU power among the alternative frameworks for power saving, primarily because of its epoch based design. Rubik outperforms Pegasus most of the time. On the other hand, Gemini leverages a two-step DVFS equipped with a NN based latency predictor for the initial frequency selection and a second predictor for the boosting time determination to improve the CPU power savings. The results in Fig. 6.12 (a), (b) and (c) show that Gemini consumes the least amount of CPU power, across all three traces. We show the average power savings in Fig. 6.12 (d). Rubik achieves around 23.7%-27.8% CPU power savings with the three traces. The average power saving for Pegasus on the three traces is in the range of 20.07% to 24.72%. The best power management among these, Gemini, achieves up to 42.15% power saving on Lucene trace which is 1.53 times better than Rubik and outperforms Pegasus by 70.5%.

In addition to the power savings, the request latency distribution for each scheme on the Wikipedia trace is shown in Fig. 6.13 (a). Rubik, Pegasus and Gemini save CPU power because they can shift the "knee" of latency distribution to the right, towards the deadline. Although the tails of their latency are similar in Fig. 6.13 (a), Gemini is able to slow down more requests, such that they are closer to the deadline. This allows Gemini to have better energy savings. The detailed tail latency and deadline violation rates for this trace-driven experiment are presented in Fig. 6.13 (b). On the top sub-figure, the baseline has the smallest 95th tail latency with 13.8 ms (much smaller compared to our 40ms time budget). Rubik utilizes this latency slack to shift the tail to 37.9 ms, completing just before the deadline, to save energy. The 95th tail latency on Gemini is 39.3 ms. However, Pegasus has 95th tail latency of 44.2 ms, thereby resulting in more than 5.8% (highest) of the requests

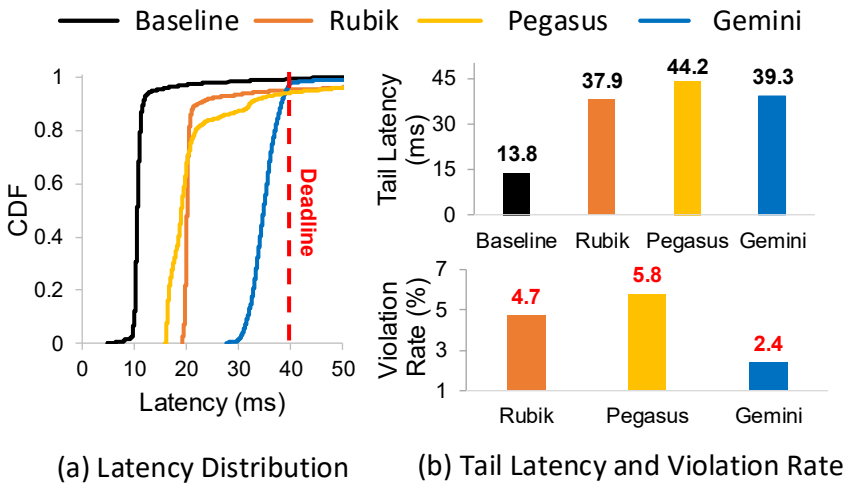


Figure 6.13: Gemini achieves the smallest tail latency and lowest deadline violation rate.

having deadline violations as seen in the bottom sub-figure. Rubik always uses the 95th percentile on the service time distribution for frequency selection. Thus, Rubik's deadline violation rate is lower, at 4.7%. But, by having a precise per query service time prediction and using a second error predictor to reduce the deadline violation, Gemini achieves a 2.4% deadline violation rate which is less than half of that for Pegasus. Thus, Gemini achieves a much better balance between the CPU power saving and deadline violation rate than the alternatives we compare.

6.6.4 Breakdown of Power Saving

In Gemini, our two-step DVFS (for limiting the frequency transition overheads) initially selects a low CPU frequency to save power, but incorporates mechanisms to make sure that the initial selected frequency is not too low to miss the deadline. Then, our two-step DVFS boosts the initial frequency at just the right time, such that the deadline is met. In fact, PACE [70] theoretically proves that a step-wise DVFS scheme produces

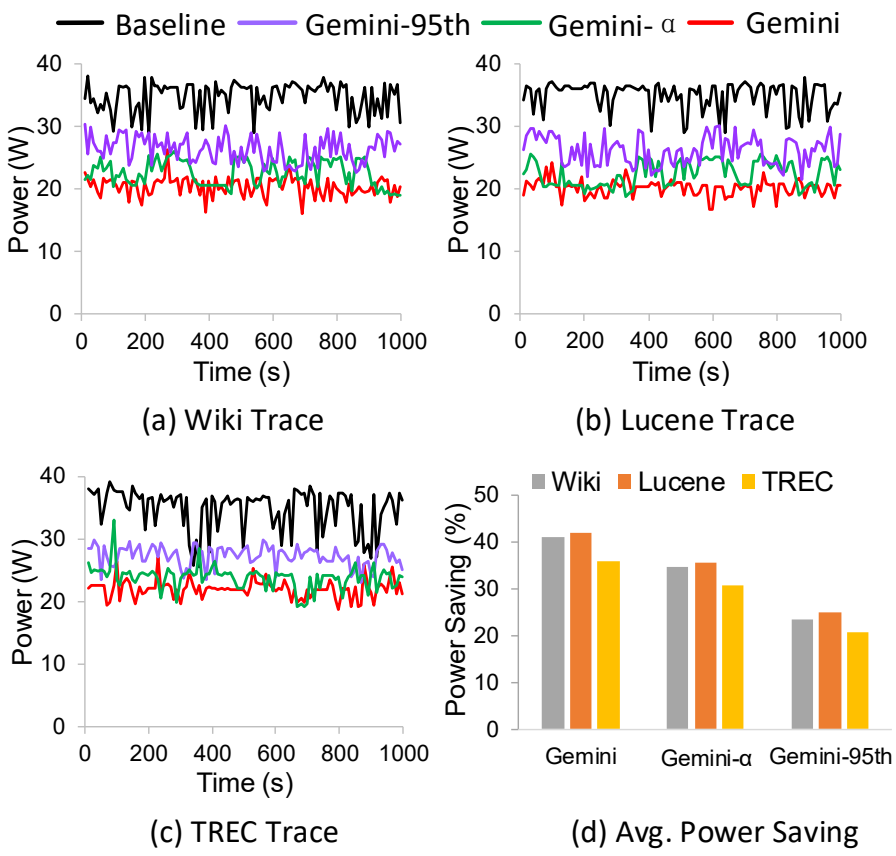


Figure 6.14: Comparing Gemini, Gemini-α, and Gemini-95th.

the optimal CPU power saving when a task's total work is unknown. Both synthetic and trace driven experiments show a significant power saving for Gemini. To further enhance the power management, we develop two separate latency and error predictors with high accuracy. The latency predictor enables a suitable initial frequency selection in our two-step DVFS and the error predictor enables our scheme to boost the initial frequency at the correct time.

In this experiment, we examine the reasons behind the significant power savings of Gemini on three different query traces. We develop two variants of Gemini. In particular, we seek to determine the impact of our NN-based predictors. First, Gemini-α described

in the previous experiments is reused. Gemini-α doesn't use the second error predictor of Gemini at all. Building on Gemini-α, "Gemini-95th" also removes its latency predictor. Instead, similar to Rubik, we use the 95th percentile of service time distribution to estimate a query's latency in Gemini-95th. In Fig. 6.14 (a), (b) and (c), we report the CPU power variation with time for the Wikipedia, Lucene and TREC traces. For all the traces, the power consumption of Gemini-α is slightly higher than Gemini. This is because the moving average in Gemini-α is unable to provide a measure of each request's precise residual work. Thus, the two-step DVFS has to boost the CPU frequency earlier to achieve a lower deadline violation rate. When we further disable the latency predictor and use the 95th percentile of the service time distribution for frequency selection, Gemini-95th consumes even more CPU power than both Gemini-α and Gemini. The average power saving results are shown in Fig. 6.14 (d). When comparing Gemini with Gemini-α, we find that the use of second error predictor brings us around 6.53% more power savings on the Lucene trace. On the other hand, the power saving improvement contributed by our service time predictor in Gemini is 10.8% on the Lucene trace. On the TREC trace, the average power saving of Gemini is 36.09%. Breaking this down, when only the two-step DVFS (without the predictors) is used, as in Gemini-95th, the power saving is 58% of Gemini. By having the latency predictor but not the error predictor in Gemini-α, we achieve 86% of the complete design's power saving. Thus, we conclude that it is important to have both the two-step DVFS *and* the two predictors in Gemini for significant power savings.

Chapter 7

Coordinated Time Budget Assignment at the Aggregator

7.1 Introduction

The primary goal of this chapter is to reduce a search system's tail latency and resource consumption with negligible quality loss. To achieve this goal at the granularity of an individual query, we propose Cottage (i.e., **co**ordinated **t**ime budge**t** **a**ssi**g**nm**e**nt), a coordinated framework integrating the prediction and decision making at both the aggregator and the ISNs. An ISN's quality and latency estimations are conducted independently at each ISN server, where there is more complete information of the sharded index. An ISN's quality means the number of documents it contributes to the top-K client-side search results, whereas its latency means the time taken to process the query at the particular ISN including local queuing. Having each ISN predict its quality and latency concurrently leads

to a much more scalable design. But we recognize that the overall query latency and quality determination needs a global view of the responses from all the ISNs. Thus, we gather the ISN prediction results at the aggregator and design a centralized optimization algorithm to determine the time budget (cutoff time) for the query. The distributed decision making we perform here has little impact on the request's service time as the network latency in data centers is typically at the microsecond granularity [79].

Two separate NN models with distinct query features are developed at the ISN for quality and latency prediction. Unlike Taily [7], which assumes a Gamma score distribution for all requests, our NN model adapts to each query's unique score distribution and doesn't need to predict a global K^{th} maximal score. All our query features are based on the term statistics [72], which are calculated during the indexing phase. The prediction accuracy of our quality model is 95.7% while only taking 80 microseconds for the inference. Similarly, the latency prediction has a high accuracy of 87% with negligible overheads.

In our framework, we incorporate a frequency boosting technique to accommodate slow ISNs that contribute highly to the overall quality. The time budget determination algorithm uses four different predictions from each ISN: the quality for top-K, the quality for top-$K/2$, the latency estimate at the current frequency at an ISN and the latency using the highest (boosted) frequency. With these predictions, the aggregator first ranks the ISNs based on their predicted quality for top-K results and cuts off those ISNs with zero contribution to the top-K results. In order to consider high quality slow ISNs, the remaining ISNs are re-ranked with their boosted latencies using the highest CPU frequency. With this, the time budget is determined by the ISN with the longest boosted latency (the last response

the aggregator waits for), but with a high quality contribution to the top-$K/2$ results. We select its boosted latency as our search query's deadline. The aggregator's ability to find the right balance between quality and latency (because it has the visibility across all ISNs) enables us to find a far superior, coordinated decision compared to previous work. The evaluation results on our platform show that the integration between aggregator and ISNs bring 37.35% more latency improvement while achieving a similar search quality.

The entire framework of Cottage is implemented on the well-known Solr [101] search engine. Experimental results on two representative query traces prove that Cottage yields a 2.41 times shorter average query latency when searching 2.67 times fewer documents than exhaustive search. At the same time, we achieve a good P@10 search quality of 0.947 (out of 1). Both these results are much superior compared to CSI based Rank-S [65] or Taily [7]. Our major contributions in this chapter are the following:

- We propose a coordinated framework between the aggregator and ISNs to improve the search engine's query-specific latency and quality while improving the search efficiency.
- Two separate neural network models with distinct query features are developed to predict each ISN's quality and latency.
- We design an algorithm at the aggregator to assign a dynamic time budget for each search query, considering both the quality and latency predictions by the individual ISNs.
- The Cottage framework is implemented in a real testbed using representative query traces to prove the superiority of our technique.

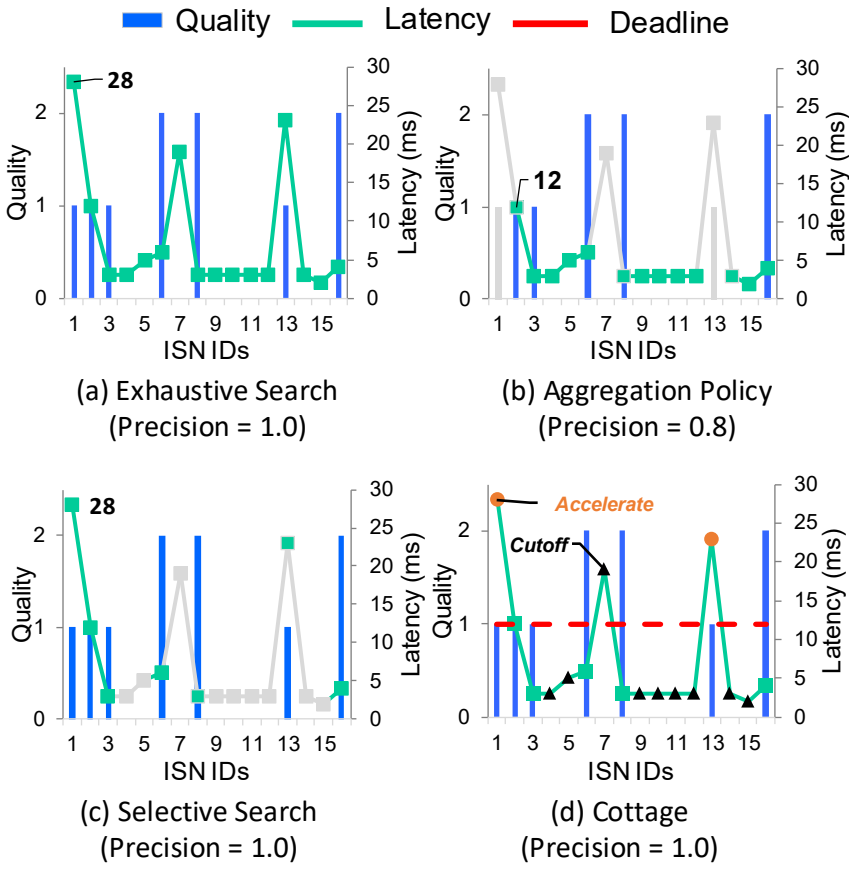


Figure 7.1: The policy comparison between exhaustive search, aggregation policy, selective search and Cottage.

7.2 Motivation

We first consider the case of exhaustive search. In Fig. 7.1 (a), we plot the quality (bar with left hand side Y-axis) and latency (line with right hand side Y-axis) results for the query "Canada" on a Solr search engine with 16 ISNs. In exhaustive search, with 100% P@10 precision [56] the time budget for the search request has to be at least 28ms such that the slowest ISN-1 can return its results before the deadline. Epoch-based aggregation policies [22, 123] find a cutoff parameter that produces an optimal latency reduction for most of the requests with an acceptable search quality. However, they consider cutting off

long-tail servers irrespective of their quality contribution. The results after applying the aggregation policy are shown in Fig. 7.1 (b), in which the stragglers (the grey ISN-1, 7, 13) on Fig. 7.1 (a) are removed. In Fig. 7.1 (b), we assume that a time budget of 12ms yields the best latency improvement for most of the queries in an epoch. If we remove ISNs-1, 13, they provide the two top-10 search results. Excluding them will deteriorate the search quality by 20% (i.e., yielding a P@10=0.8).

Let us now look at selective search frameworks [97, 104, 65], which only gather responses from a subset of ISNs to reduce the resource usage of a distributed search engine. A query's response time from an ISN is generally not considered, when selecting ISNs to use in the search. For the same example, ISN-4, 5, 7, 9, 10, 11, 12, 14, 15 in Fig. 7.1 (c) will be excluded when selective search is used. However, the overall latency of query "Canada" is not optimized even if we cutoff some low quality ISNs. Thus, it is essential to consider both the latency and quality when assigning a query's time budget at the aggregator. This motivate us to have the design of Cottage as shown in Fig. 7.1 (d). When determining the time budget, the ISN with a long latency and a high quality contribution should be retained, instead of directly dropping them off in the aggregation policy. There are many approaches such as pruning [105] or parallelization [55] on an ISN to accelerate the request processing. In this chapter, we propose the CPU frequency scaling to speed up the search request, as it doesn't hurt the search quality.

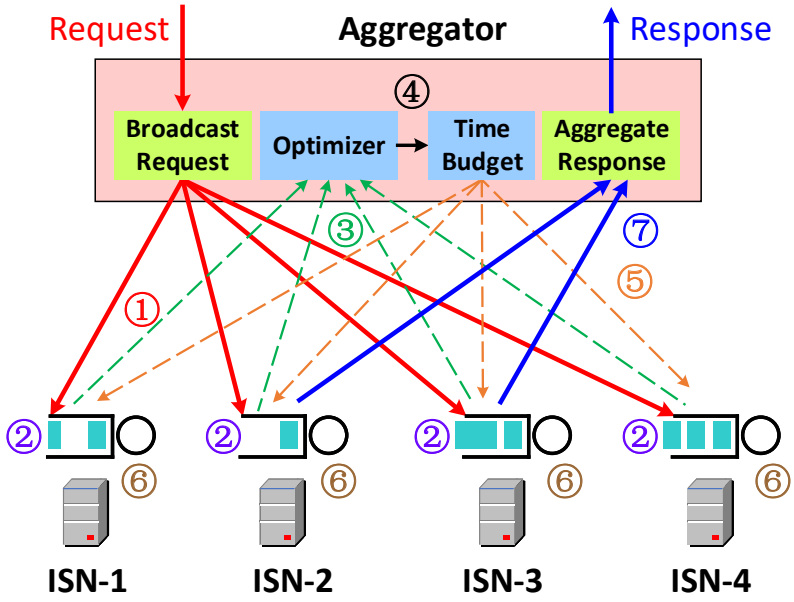


Figure 7.2: The Cottage framework needs the coordination between aggregator and ISN servers for quality prediction, latency prediction and time budget determination.

7.3 Cottage Design

7.3.1 Overview

Fig. 7.2 provides the design overview of our framework Cottage. Whenever a search request arrives at the aggregator, Cottage will first broadcast the request to all the ISNs (i.e., step 1 in the figure) as in an exhaustive search. At step 2, Cottage predicts an ISN's quality contribution to the P@10 result for a given query, based on a NN model described in section 7.3.2. Apart from the quality prediction, Cottage also needs to predict each query's service time at step 2 as both quality and latency have to be considered when determining a query-specific time budget. We use the same method descried in chapter 6 to precisely estimate each query's service time. These query-specific quality and latency predictions at the ISN are sent back to the aggregator in step 3 of Fig. 7.2. Then, Cottage

uses a centralized optimizer for latency and quality optimization (in step 4). The key point of our optimization is to assign a dynamic time budget (i.e., deadline) for each query.

The process of determining the time budget is further described in section 7.3.3. In step 5 of Fig. 7.2, the dynamic time budget for a query is sent back to all the ISNs. With the assigned time budget, the ISN will process the query accordingly in step 6. Finally, the ISN sends its responses to the aggregator in step 7. The responses from ISNs after the deadline are ignored by the aggregator. For example, in Fig. 7.2, only ISN-2 and ISN-3 will send their responses to the aggregator, since ISN-1 and ISN-4 have a low quality contribution or an excessively large predicted latency. Thus, the tail latency for the parallel request across the ISNs are reduced. The communication between the aggregator and ISN doesn't incur significant latency overhead, as the data center network round trip time can be kept low (typically a few micro seconds) [79], compared to the tens of milliseconds of service time for the search application.

7.3.2 Quality Prediction

State-of-the-art shard ranking algorithms [104, 65, 97] are centralized designs at the aggregator. The only distributed scheme we are aware of in the literature, Taily [7], assumes a Gamma distribution for a query's scores against all the relevant documents at an ISN. Instead, we develop a novel NN model with carefully selected features to predict each query's quality contribution on an ISN. With better prediction accuracy, Cottage avoids falsely dropping an ISN's response that will eventually contribute to the final P@10 result to a client.

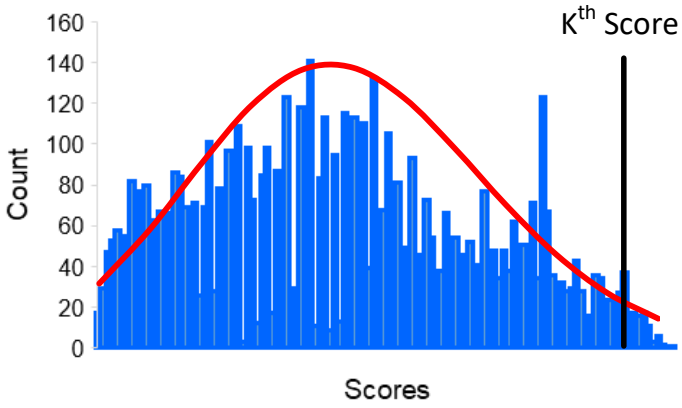


Figure 7.3: Histogram of query scores and its fitted Gamma distribution.

Let us assume that the aggregator of a search engine sends the top-K final results to the client. Then, an ISN's quality is defined by the number of documents it reports that will be included in the final top-K results. If we can infer a search query's dynamic score histogram, it is then easy to predict an ISN's quality on a per-query basis. In Fig. 7.3, we present the histogram (i.e., blue bars) of relevant scores for a specific query on ISN-1. The documents without any relevant query terms are ignored. This observation motivates the design of Taily [7] to dynamically predict a query's score distribution by assuming that the score distribution follows a Gamma distribution [58, 7]. During runtime, they predict the parameters of the Gamma distribution according to static query term statistics which are obtained during the indexing phase. However, a query's scores typically do not perfectly fit a Gamma distribution, thus resulting in an inaccurate ISN cutoff. In Fig. 7.3, we also plot the fitted Gamma distribution (i.e., red line). We can observe that the $P(X > K^{th})$ from the Gamma distribution is not quite the same as the distribution shown in the histogram. If we improperly cutoff some ISNs that will significantly contribute to the top-K results, search quality will suffer.

Table 7.1: Features for Quality Prediction

Feature Name	Example for "Tokyo"
First quartile score	2.46
Arithmetic average score	4.88
Median score	7.16
Geometric average score	3.91
Harmonic average score	2.2
Third quartile score	4.72
K^{th} score	11.08
Max score	14.46
Score variance	8.22
Posting list length	5975

Cottage proposes a NN model to predict each query's quality contribution. The output of our NN model is the number of documents at an ISN that will be included in the corresponding top-K results. In Table 7.1, we list all the features used in our NN model. Our design is based on the premise that we can utilize the query's aggregated statistics, such as the arithmetic average score and max. score, to capture the score distribution. By training the model with a large amount of observed samples from the past, we can accurately predict a query's quality when observing similar score distributions. The different score percentiles, as listed in Table 7.1 row 2 to 10, can be easily obtained from index term statistics [72, 61, 55]. Although two queries might have the same score distribution, their quality contribution might be different as the number of documents they have to search are different. Thus, the posting list length (i.e., document count) becomes our last, but very important, query feature in Table 7.1.

For model training, we select a NN model with 5-hidden layers as it maintains a good balance between accuracy and inference time. Each hidden layer has 128 neurons and uses the ReLU activation function. The model is trained by the Adam optimization

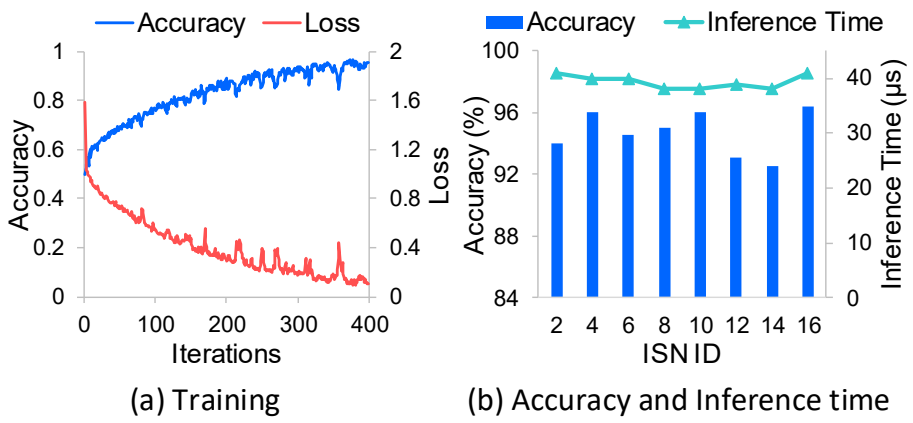


Figure 7.4: The prediction accuracy and inference time for quality prediction.

algorithm [63] with sparse categorical cross-entropy loss function. In Fig. 7.4, we present the prediction accuracy and inference time of our quality predictor. Based on the query features listed in Table 7.1, we show that the accuracy of quality prediction (left-hand side Y-axis of Fig. 7.4 (a)) improves when we train the model over more iterations to reduce the value of loss function (right hand side Y-axis of the same figure). The prediction accuracy of our model can be up to 95.7%. We reach a point of diminishing improvement after 600 training iterations. Next, we compare the quality predictor's accuracy and inference time on various ISNs. Each ISN has a separate NN model trained with its own index data. In Fig. 7.4 (b), our quality predictor achieves an average of 94.71% accuracy (left-hand side Y-axis) across ISNs. In addition to the prediction accuracy, the inference time for each query is another important performance metric. The inference time overhead is shown by the right hand side Y-axis of the same figure. Our quality predictors only incur at most 41 microseconds of inference time, compared with tens of milliseconds query's service time [55].

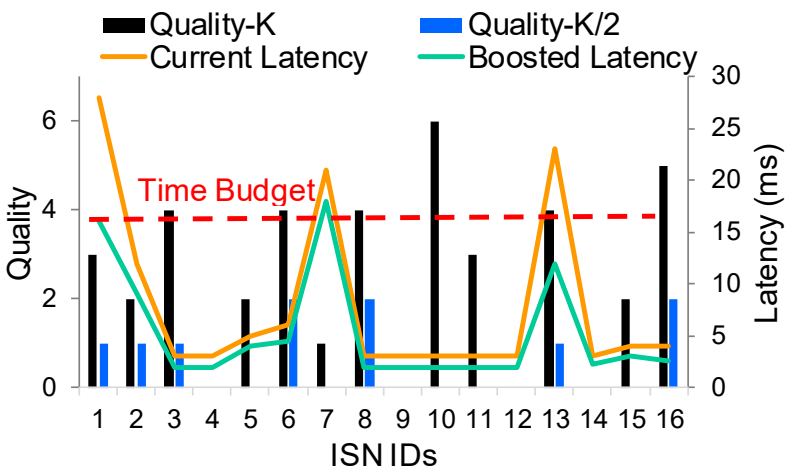


Figure 7.5: An example (with $K = 20$) for time budget determination in Cottage.

7.3.3 Time Budget Determination

The optimizer for time budget determination in Cottage needs two predictors: the quality contribution prediction and the latency prediction. As shown in Fig. 7.5, both the Quality-K (black bar) and Quality-$K/2$ (blue bar), (both with left-hand side Y-axis) are used by our algorithm. The Quality-$K/2$ prediction means the number of documents that an ISN will contribute to the top-$K/2$ client-side results. Similarly, Cottage obtains each ISN's predicted latency under the current CPU frequency f (yellow line) and the highest CPU frequency (green line), (both with right-hand side Y-axis). The boosted latency is the shortest time that an ISN server can finish a request through frequency boosting. With the aforementioned quality and latency predictions, our algorithm dynamically assigns a minimal time budget to the parallel search requests such that the latency and search efficiency are optimized with negligible quality loss.

The details of the time budget determination algorithm is shown in Algorithm 1. I

Algorithm 1: Time Budget Determination

```
1  I: set of ISNs associated with quality and latency predictions
     < Q^K, Q^{K/2}, L^{current}, L^{boosted} >
2  T: time budget
3  j = 0, N = I.size()
4  while j < N do
5      if I_j.Q^K equals 0 then
6          drop ISN I_j
7          remove I_j from I
8      end
9      j=j+1
10 end
11 DescSort(I, L^{boosted})
12 T = I_0.L^{boosted}, j = 0, N = I.size()
13 while j < N do
14     if I_j.Q^{K/2} ≠ 0 then
15         T = I_j.L^{boosted}
16         break
17     end
18     j=j+1
19 end
```

is the set of ISNs for a search engine. Each ISN I_j has four prediction results: the quality-K Q^K, quality-$K/2$ $Q^{K/2}$, latency under current frequency $L^{current}$ and latency under highest frequency $L^{boosted}$. In the first stage of the algorithm (line 3-10), all the ISNs are ranked by the Quality-K predictions. To improve the search efficiency, the ISNs with zero Quality-K are cut off and removed from the ISN set I. Specifically, the ISN-4, 9, 12, 14 on the example of Fig. 7.5 are cut off. In line 11 of Algorithm 1, Cottage re-ranks the remaining ISNs by the descending order of boosted latency because we want to find the shortest time budget to meet quality constraints. In Fig. 7.5, the re-sorted ISN list is <7, 1, 13, 2, 6, 5, 15, 16, 3, 8, 10, 11>. Then, the lines 12-19 of Algorithm 1 try every ISN's boosted latency as the time budget from the beginning to the end of ISN set I, until an ISN j has a quality contribution to the top-$K/2$ results. We select ISN j's boosted latency as the final time budget T.

To make it clear, an example is given in Fig. 7.5. First, the initial time budget is determined by ISN-7's boosted latency of 18 milliseconds. However, most of the remaining ISN's current latency and boosted latency are far below the 18 milliseconds time budget. We notice that the ISN-7 doesn't contribute any documents to the most important top-$K/2$ results [56]. It is reasonable to trade off a little bit of the bottom $K/2$ result's quality for reduced response time. Thus, Cottage will choose a shorter time budget and drop ISN-7. Next, we try ISN-1's boosted latency of 16 milliseconds. As ISN-1 contributes one document to the most important top-$K/2$ results, we have to retain the response of this ISN and select a time budget of 16 milliseconds. Finally, the time budget line in Fig. 7.5 will stop moving towards the X-axis due to the quality constraint of ISN-1.

Compared with the previous selective search schemes [104, 65, 81] which only consider the quality contributions, the Cottage algorithm reduces the tail latency to 16 ms, compared to 28 ms achieved with the selective search. On the other hand, existing aggregation polices [123, 22, 53] cut off ISN-1 because of its higher search latency, thus ignoring its significant quality contribution. Our algorithm achieves a proper balance between the search quality and latency. After determining the time budget, ISN-1 and ISN-13 will boost their current CPU frequency as their predicted latency at the current frequency is longer than the given time budget.

7.4 Implementation

Cottage is implemented on the well known Solr search engine [101]. In Solr, the application instance can work as the aggregator or an ISN server. On a search request arrival, if there are multiple destination shards, the Solr instance works as an aggregator and handles the search request as a distributed request to multiple ISNs. Our centralized optimizer is implemented after the aggregator broadcasts the distributed request to all its destinations. As the Solr search engine already has multiple rounds of communications between the aggregator and ISNs which are achieved by the concept of query component, the gathering of quality and latency predictions in Cottage is implemented by adding an additional query component as part of minimizing the overhead. If a search request's destination matches a Solr instance's ID, then it is viewed as a local request and the Solr instance works as an ISN server. At the ISN, our logic in Cottage is implemented before the ISN begins to process the query. In Cottage, we use the Keras API of TensorFlow [1] to

achieve the NN prediction models. For the request boosting, the Advanced Configuration and Power Interface (ACPI) is utilized to update the CPU core's frequency at runtime. Our experimental setup has two machines connected by a 1G Ethernet link: one as the client and the other as the search engine server. The server machine is a 24 core Intel Xeon E5-2697 CPU, 128G memory running CentOS 7 operating system. On the client side, we wrote a Python program to replay our real search query traces. Two representative query traces are used in our experiments: the Wikipedia [107] and the Lucene nightly benchmark [48]. On the server side, we deploy a 16 ISNs Solr search engine. In the search engine, we index the complete dump of entire English Wikipedia web pages on December 1st, 2018. This 65GB index has a total of 34 millions documents. The search engine machine supports per core frequency scaling. The CPU frequency can be selected in the range of 1.2 GHz to 2.7 GHz.

7.5 Evaluation Results

We compare Cottage with the baseline policy of exhaustive search as well as the state-of-the-art schemes such as Rank-S [65] and Taily [7]. For the baseline, a search request will be executed on every ISN of the distributed search engine. In exhaustive search, the aggregator sends search responses to the client only when it receives the response from the slowest ISN. Rank-S is a centralized design in which each ISN's quality contribution is estimated by using a CSI. In our experiments, every ISN's index is sampled by 1% to form the CSI at the aggregator. Rank-S uses the fixed threshold for all requests to cutoff low quality ISNs. Cottage is also compared with a distributed design, Taily. The major feature

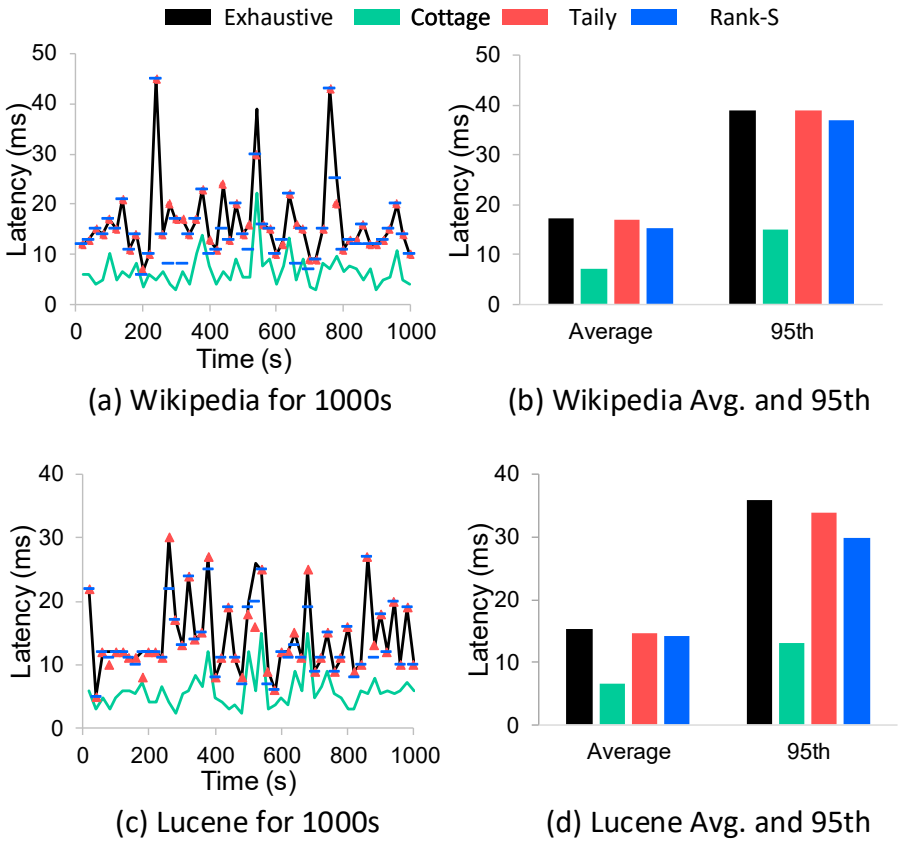


Figure 7.6: On both query traces, Cottage reduces the average client-side latency by 54%.

of Taily is that it uses a Gamma distribution for the scores to infer each ISN's quality contribution.

7.5.1 Overall Latency

Fig. 7.6 shows the overall latency for requests from the Wikipedia and Lucene query traces run for 1000 seconds. In Fig. 7.6 (a) for the Wikipedia trace, the requests' overall latency (black line) using exhaustive search varies in the range of 4ms to 65ms. For Taily (red dot), its overall latency in Fig. 7.6 (a) is similar to exhaustive search most of the time. This is because it only excludes the ISNs that have zero contribution to the P@10

results, without considering the dimension of latency. Sometimes, a low quality ISNs may happen to have a long latency (e.g., the query made around the 780th second). In such a case, Taily can improve a search request's overall latency. We plot the corresponding average and 95th tail latencies in Fig. 7.6 (b). As shown in Fig. 7.6 (b), Taily improves the average request latency of exhaustive search only by 1.16% and reduces the 95th tail latency by 1.2%.

Rank-S (blue line) performs better than Taily in Fig. 7.6 (a) and (b). On average, it reduces the request's overall latency by 11.12%, from 17.26ms in exhaustive search to 15.34ms. The improvement of 95th tail latency is similar. Due to the sampling design in Rank-S, we only know the relative importance between ISNs without any knowledge regarding their contributions to the P@10 results. Rank-S may wrongly cutoff more ISNs and thus produce a better overall latency, but at the cost of quality. Finally, our design Cottage results in the shortest request latency in Fig. 7.6 (a) and (b). In Fig. 7.6 (a), Cottage (green line) outperforms the baseline exhaustive search and the other frameworks compared all the time. As presented in Fig. 7.6 (b), the average latency of requests on the Wikipedia trace is reduced by 54% compared with the exhaustive search. What is more, we improve the request's 95th tail latency by 2.6 times, from 39ms in exhaustive search to 15ms. In Fig. 7.6 (c) and (d), we plot the latency results on the Lucene query trace. Similarly, Cottage has a 2.29 times better average latency and 2.74 times better 95th tail latency than exhaustive search, respectively. The major reason for our latency reduction is due to cutting off the latency tail when it has no quality contribution and accelerating the ISNs that have a long latency but have a high quality contribution.

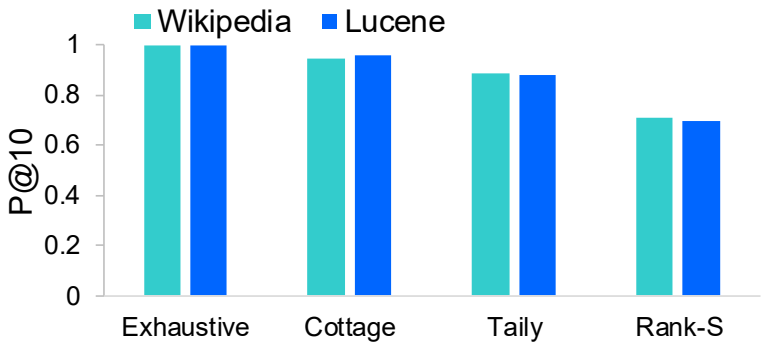


Figure 7.7: The average P@10 search quality results on Wikipedia and Lucene query traces.

7.5.2 P@10 Quality

Besides the request's latency, the P@10 quality is another very important performance metric for a search engine. Under the same experiment setup, we measure the search requests' P@10 quality for both the Wikipedia and Lucene traces. The average P@10 quality results for 1000 seconds are reported in Fig. 7.7. With exhaustive search, the precision of the search results is always 1, as every document in the entire data collection will be retrieved. With an accurate per query quality prediction, Cottage achieves an average of 0.947 P@10 search quality on the Wikipedia trace as shown in Fig. 7.7. Similarly, the P@10 quality on the Lucene trace is 0.955. We sacrifice the P@10 search quality a little bit because Cottage drops the ISNs with low quality contributions but having an extremely long latency for improved search response times. It is reasonable to trade-off around 5% of the search quality for at least 2.17 times improvement in search latency. On the same figure, Taily has a search quality of 0.887 on the Wikipedia trace and 0.878 P@10 quality on the Lucene trace. The P@10 quality results of Cottage are at least 6% better than that on Taily. As Taily utilizes a Gamma distribution to infer each ISN's quality contribution,

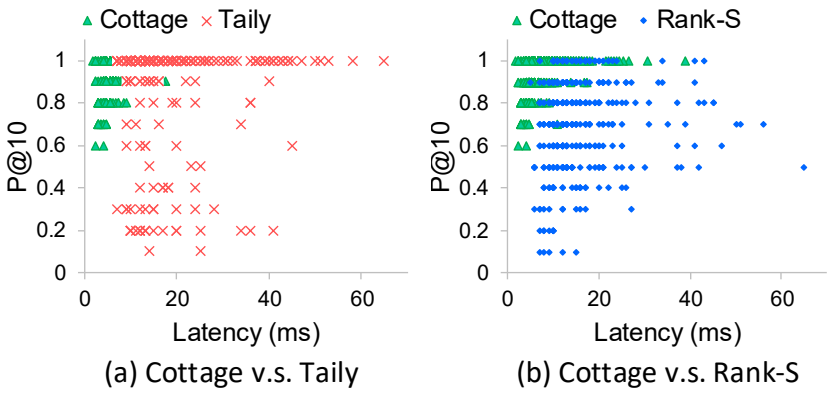


Figure 7.8: Latency and quality distribution of Cottage in Wikipedia trace.

it is difficult for it to have a perfect prediction and thus a good search quality.

Across all the frameworks compared, Rank-S has the worst search quality due to its sampling design at the aggregator. In Rank-S, we only have the relative rankings between ISNs based on the centralized index samples. It is inevitable that cutoff of an ISN can be imperfect. The average P@10 quality of Rank-S is at most 0.709 in Fig. 7.7, which is 25.13% worse than Cottage on the same trace. Next, we plot the latency and P@10 quality results together in Fig. 7.8. Every dot in the two figures represent one query from the Wikipedia query trace. In Fig. 7.7 (a) and (b), we observe that most of the queries on Cottage (green dots) stay at the top-left of the figure, which means that Cottage keeps a good search quality while keeping the overall latency small. However, the queries of Taily (red dots) in Fig. 7.8 (a) and the queries of Rank-S (blue dots) in Fig. 7.8 (b) scatter across the entire range of quality. Their optimizations are at the cost of poorer search quality.

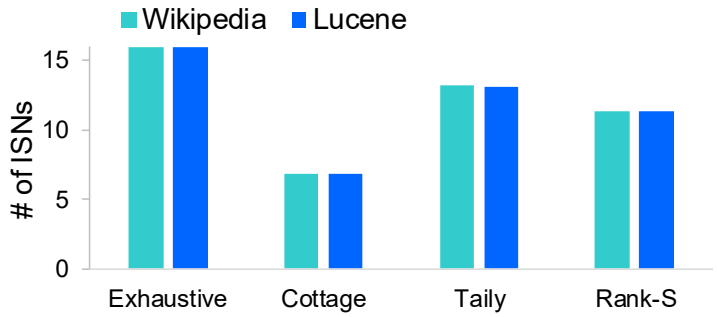


Figure 7.9: The average number of selected ISNs for a query.

7.5.3 Active ISNs

We now evaluate the number of active ISNs for a query after cutting off the slow ISNs and those not contributing to the query's search quality. Fewer active ISNs means better resource usage and search efficiency. Fig. 7.9 shows the average number of ISNs selected for a search request using 1000 seconds of the Wikipedia and Lucene traces. Since exhaustive search doesn't cut off any ISNs, its selected number of ISNs is always 16 (i.e., all the ISNs in our experimental setup). On both query traces, Cottage only needs at most 6.81 out of 16 ISNs in Fig. 7.9 to achieve a search quality of 0.947 as shown in Fig. 7.7. Our scheme enables the search engine to use a minimal number of ISNs for a high quality search result. As a comparison, Taily retrieves results from an average of 13 ISNs. Our scheme, Cottage, only needs results from almost 7 fewer ISNs than Taily (Fig. 7.9), but has 8% better P@10 quality than Taily, as seen in Fig. 7.7. This is because of the more accurate neural network based quality prediction. Similarly, the number of selected ISNs on Rank-S is around 11 on both query traces, which is 61% larger than that of Cottage.

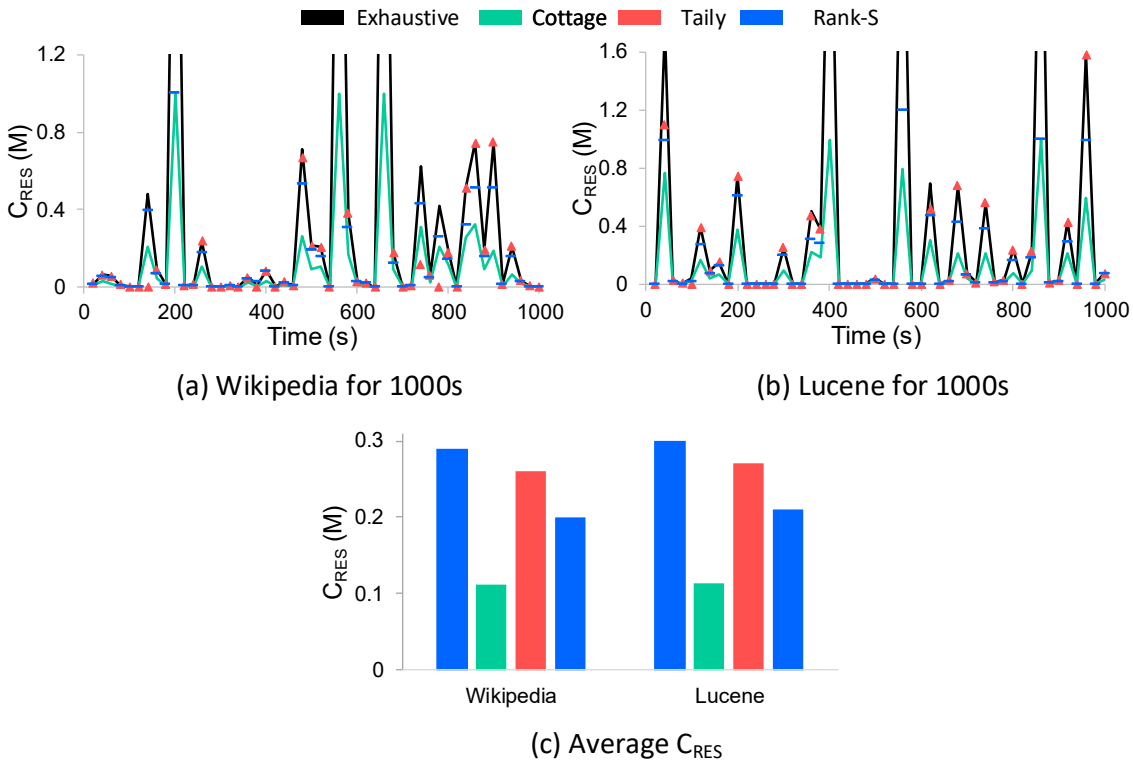


Figure 7.10: The number of searched documents for exhaustive search, Taily, Rank-S and our Cottage.

7.5.4 Document Efficiency

Similar to previous research [65, 81, 7], we utilize the performance metric of C_{RES} [7] to quantify the efficiency of a search engine. C_{RES} is the number of documents across all the used ISNs to search for the top-10 results for a given query (fewer the better). Rank-S also considers the number of scored documents in the CSI. In Fig. 7.10 (a), the number of searched documents (i.e., C_{RES}) with exhaustive search (black line) varies between 2K to 5.4M across different Wikipedia search queries. The corresponding average value of C_{RES} is given in Fig. 7.10 (c). By cutting off the low quality ISNs, Taily (red dots) in Fig. 7.10 (a) reduces the number of searched documents only on a limited number of data points

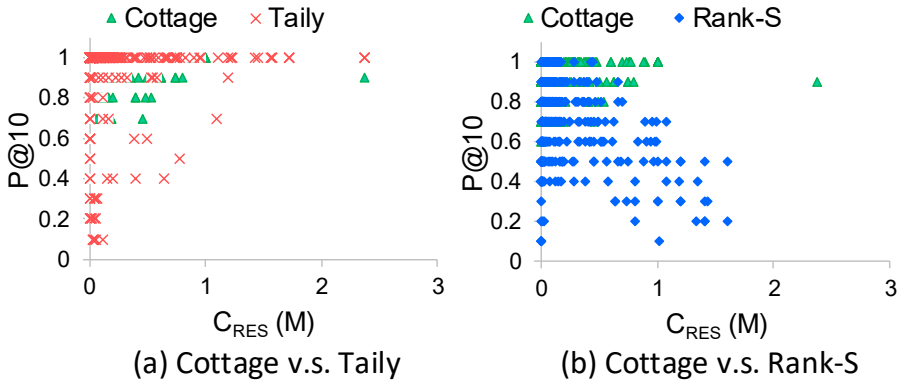


Figure 7.11: The resource reductions in Taily and Rank-S are at the cost of worse search qualities.

(e.g., the 740th second). Accordingly, its average value of C_{RES} (0.26M) in Fig. 7.10 (c) doesn't improve too much compared with the exhaustive search (0.299M). The performance of Rank-S (blue line) is similar to Taily in Fig. 7.10 (a). On average, the number of searched documents in Rank-S is 0.2M for the Wikipedia trace. Finally, Cottage has the smallest value of C_{RES} most of time, as in Fig. 7.10 (a). In Fig. 7.10 (c), Cottage retrieves an average of 0.11M documents for the queries using the Wikipedia trace, which is 2.67 times less than the baseline exhaustive search, and 1.8 times better than Rank-S. As shown in Fig. 7.10 (b), the comparison results of C_{RES} are similar with the Lucene trace. We improve a search engine's efficiency (in terms of R_{RES}) by 265% on the Lucene trace, compared with exhaustive search. Additionally, the average C_{RES} of Cottage is just 54.5% of Rank-S on the same Lucene trace.

In Fig. 7.11, we show the trade-off between C_{RES} and the search quality. In each sub-figure, each dot is one query from the Wikipedia trace with its C_{RES} value shown by the X-axis and the P@10 quality is in the Y-axis. The queries for Cottage (green triangles at top-left of each figure) retrieve a smaller number of documents with a good search quality.

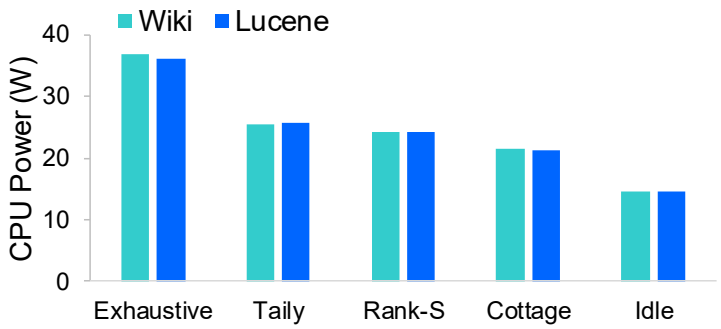


Figure 7.12: Average power consumption on the Wiki and Lucene traces.

On the other hand, frameworks such as Taily in Fig. 7.11 (a) and Rank-S in Fig. 7.11 (b) have many dots with a low search quality, even though they search more documents. Cottage achieves a good balance between search efficiency and search quality.

7.5.5 Energy Efficiency

With fewer ISNs and documents searched, the power consumption of the entire search engine per query will reduce. Fig. 7.12 compares the power consumption of the different approaches, as well as the case when the search engine is idle. As all the 16 ISNs of our Solr search engine are deployed on the same physical server, we measure the CPU chip's package power (including the L1 cache) on different ISN selection schemes to compare their energy efficiency. The CPU power is measured by using the Intel Running Average Power Limit (RAPL) interface, which reads the counters on the CPU sensor. As shown in Fig. 7.12, the power consumption of exhaustive search is around 36W for both traces, as all the ISNs are active throughout the 1000 second experiment. Taily reduces the CPU power to around 25W because it cuts off some low quality ISNs. On average, it saves a search engine's power consumption by 31.12% compared with the baseline exhaustive

search. Similarly, Rank-S has an average of 24W CPU power consumption which is around 66% of exhaustive search. Finally, we can observe that Cottage consumes the least amount of CPU power among all the compared approaches. It has an average of only 21W power consumption. The power saving of our approach compared with exhaustive search is 41.3%, a significant reduction because the platform's idle power is already 14.53W. This is mainly because Cottage needs the least number of active ISNs and the fewest searched documents for a good quality result.

7.5.6 Impact of Different Components

We now show how the accurate predictions and coordinated design (between the aggregator and ISNs) in Cottage contribute to its significant improvement in search latency, quality and efficiency. To achieve this goal, two variants of Cottage are implemented. The first variant, Cottage-without ML, utilizes the Gamma distribution based prediction of Taily to estimate each ISN's quality contribution, instead of using the Machine Learning (ML) model. By doing this, the importance of accurate quality prediction can be quantified. Then, the second variant Cottage-ISN removes the integration of the aggregator and ISNs. In Cottage-ISN, we let each ISN make the optimization decision independently, without the global visibility from the aggregator. Comparing Cottage-ISN with the complete Cottage, the impact of our coordinated design between the aggregator and ISNs can be evaluated.

With the same Wikipedia and Lucene query traces, we plot the latency, quality, active ISNs and searched document comparison results in Fig. 7.13. Besides the Cottage variants, the results for exhaustive search and Taily are also presented on the same

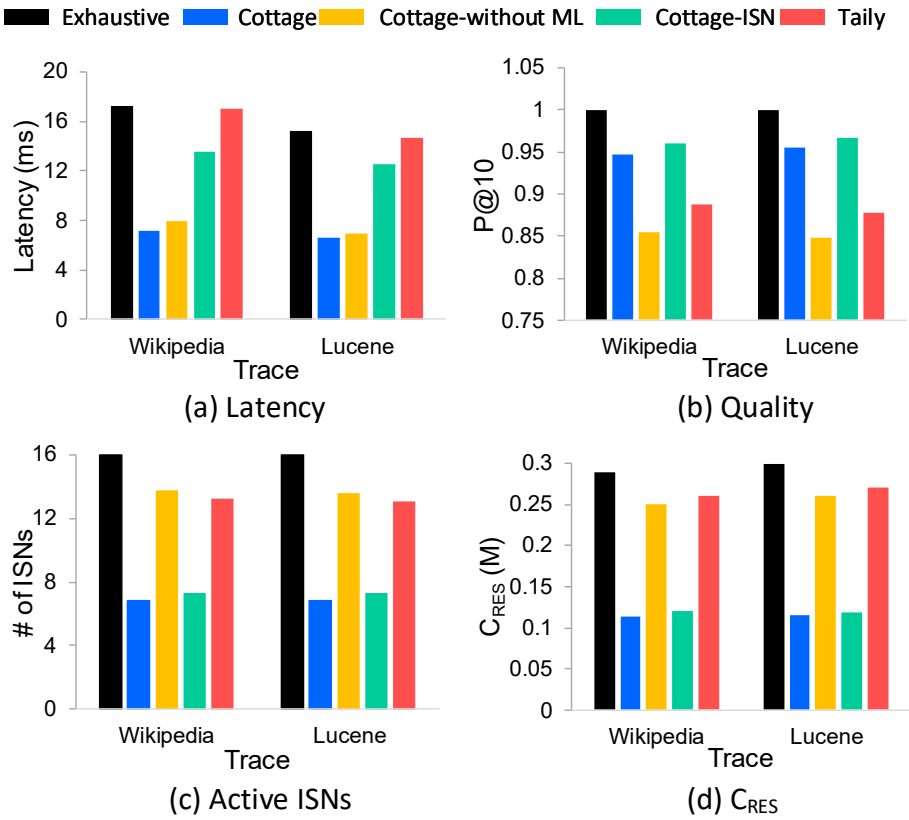


Figure 7.13: The impacts of Machine Learning (ML) based predictions and coordinated design in Cottage.

figure. As shown in Fig. 7.13 (a), the average query latencies are reduced to 12-13ms for Cottage-ISN compared with the 15-17ms average latencies for the baseline exhaustive search. Although the average latency of Cottage-ISN is better than that of Taily, it has 1.9 times longer latency than the complete design of Cottage which exploits the coordination between the aggregator and ISNs. This proves that the coordinated design in Cottage significantly reduces a search query's latency. By comparing Cottage with Cottage-without ML, we find that the more accurate quality prediction in Cottage also contributes slightly to the latency improvement, as it reduces the latency further by about 0.8ms compared with the Cottage-without ML.

Next, we show the P@10 quality results in Fig. 7.13 (b). Having a precise quality prediction, the P@10 qualities in Cottage and Cottage-ISN are as high as 0.947-0.967. However, the search quality deteriorates to around 0.85 if we use an inaccurate distribution-based prediction for the quality contribution, as in Cottage-without ML. It is essential to have an accurate neural network model for quality predictions when improving a search engine's efficiency. Finally, we compare the resource usage of different schemes, in terms of the active ISNs (Fig. 7.13 (c)) and searched documents (Fig. 7.13 (d)). In Cottage-without ML and Taily, the number of selected ISNs for a query is around 13 and the corresponding searched documents are 0.25-0.27M. When we have a ML model in Cottage and Cottage-ISN for better predictions, the resource usage for a query significantly reduces. Fig. 7.10 (c) and (d) show that the accurate quality prediction in Cottage produces 43% additional reduction in active ISNs, as well as a 48% smaller value for C_{RES}.

Chapter 8

Conclusion and Future Work

Data centers consume significant amounts of energy, costing the data center operator millions of dollars per year. Recent studies have demonstrated that a significant portion of that energy consumption can be saved by using load consolidation and application scheduling techniques. However, state-of-the-art power management frameworks for data centers have poor responsiveness to traffic variations and produce high deadline violations for latency critical applications. In this dissertation, we overcome the shortcomings of existing frameworks by proposing novel techniques to improve the energy efficiency of a data center, while reducing both latency and violation rate.

First, we find that tasks in data centers should be better allocated to servers in groups due to the complicated inter-task communications. In chapter 3, we propose Goldilocks, a holistic framework that solves the complex 'right sizing' provisioning problem in containerized data centers. A novel graph-based locality aware container placement scheme is developed, yielding minimal power consumption and task completion time. We

found that current servers are more power efficient when operating at the peak energy efficiency point. That leaves adequate headroom for burstiness in the workload. Our testbed implementation on the Twitter content caching workload proves that Goldilocks saves up to 22.7% of the entire data center's power. The energy per request in Goldilocks is 3 times better than RC-Informed, which consumes the least energy per request among the alternatives. Because of Goldilocks's locality aware placement and large headroom for load spikes, it has at least 2.56 times better task completion time than others. Simulations of a large scale data center environment also shows that our power and performance improvements are still achieved at scale.

Then, in chapter 4, we considered traffic consolidation in DCNs. State-of-the-art centralized traffic consolidation approaches have poor responsiveness to traffic variation in a DCN, especially for bursty traffic. The poor responsiveness results in much poorer energy savings, as well as higher packet drop rates and packet delays. We designed a distributed energy-aware traffic management framework, DREAM, that consolidates traffic to a minimal portion of DCN at the flowlet granularity. Flowlet scheduling produces less fragmenting of the link capacity and allows for better energy savings. The distributed agent is implemented in Open vSwitch and utilizes ECN to sense the queue build-up at the intermediate switches. Testbed results using the Wikipedia trace and the Facebook MapReduce trace prove that DREAM on average saves at least 15.8% DCN energy, compared to existing schemes such as CARPO which saves only 11.6% and ElasticTree which saves 8.4% energy. The packet drop ratio in DREAM is less than 0.01% while the best among the alternatives, ElasticTree, has 0.19% drop ratio on Facebook trace and 0.85% drop ratio on Wikipedia trace. For

application-level latency, DREAM achieves at least 30% shorter latency compared to the alternatives.

In chapter 5, we design EPRONS, a novel joint energy proportional technique for the entire data center, including both servers and the DCN. Prior DVFS or sleep states techniques concentrated on power saving in the servers without considering the DCN. EPRONS minimizes the entire data center's power consumption through dynamically searching the optimal parameter K in our linear programming model while guaranteeing the latency constraints at both DCNs and servers. Furthermore, the server part of EPRONS outperforms the existing state-of-the-art energy proportional techniques. EPRONS can save as much as 31.25% at the peak, and 25% on average, for the data center's total power budget for search workloads.

Next, in chapter 6, we concentrate on the most latency critical search application and develop a DVFS technique to improve its performance. We notice that it is difficult to predict a search request's latency perfectly that results in lower power savings and high violation rates. To save power and meet strict latency constraints, we present Gemini that employs a two-step DVFS policy for CPU frequency control. Our two-step DVFS utilizes a request's service time to initially slow down the query processing based on a precise neural network latency predictor. In the second step of our DVFS scheme, we boost the CPU frequency at a particular time to meet a request's deadline. A separate error predictor is developed in Gemini to determine the correct time for frequency boosting. We have shown that the overhead due to neural network based predictors is negligible. After implementing Gemini in the well known Solr search engine, experimental results on the Wikipedia, Lucene

nightly benchmark and TREC query traces all show that our framework can achieve up to 41% CPU power saving and outperform other state-of-the-art schemes, while reducing the deadline violation rates.

Finally, we present Cottage in chapter 7, which is a coordinated framework between the aggregator and ISNs for latency and quality optimization in distributed search. The major feature of our design is the proper partitioning of the optimization between the individual ISNs with full index information and the aggregator which has a global visibility. Cottage employs two separate neural network models with high prediction accuracy to further enhance the benefits of the aggregator and ISN coordination. With these prediction results, our optimization algorithm considers each ISN's quality contribution and latency, while achieving a good balance between the search latency, quality and efficiency. Implementation results with real query traces show that Cottage can reduce the average query latency by 54% while searching nearly 2.67 times fewer documents than exhaustive search, and still achieve a good P@10 search quality of 0.947. The results are also compared with two state-of-the-art techniques to show Cottage reduces the latency by 58% while improving the quality by 6.76%.

8.1 Future Work

Some potential future research directions are as follows:

- *Failure Resilience Aware Container Placement.* In a data center, there are different fault domains for the sake of service surveillance. Individual micro service of an application might be unavailable because of server failure, switch failure or power supply

failure. The faulty situation is usually handled through providing replicas of servers or database that can be called to service. To extend Goldilocks described in chapter 3, we will place the replicas of the same service on different fault domains by giving negative edge weights for replica-replica or replica-primary edges in the container graph. Using the min-cut graph partition algorithm, where the communication between vertices are positive in the container graph, the replicas with negative edge weights are thus partitioned into different container groups. Different container groups with the replicas are assigned to different fault domains in the DCN.

- *Minimizing Migration Costs via Incremental Graph Partitioning.* The placement of containers from one execution to the next should be stable because of concerns on migration costs such as application freeze time, migration traffic and the task startup latency. Goldilocks in chapter 3 is an epoch based scheduling system. The number of container migrations is the 'difference' between prior container grouping results and the current grouping results. In order to reduce the migration cost, we can leverage the incremental graph partitioning algorithm [88] to trade off the partitioning quality *and* the number of vertice migrations needed from old partition to the new partition.

- *Distributed Deadline Aware Traffic Consolidation.* The current distributed agents in DREAM consolidate traffic to the left-most part of a DCN topology without considering each network flow's deadline. For the urgent network flow, we can meet its deadline by dynamically increasing the path selection probability of the right-most path. Then, more flowlets of the urgent flow will be routed along the least utilized path with shorter network delays.

- *Neural Network based DVFS for Multicore Processors.* In chapter 6, we develop a neural network based two-step DVFS for the single core CPU. With multiple CPU cores, we can maintain a separate queue for each core and have a global broker to distribute the incoming requests to each core, as suggested in [72]. Each core will manage its power consumption independently by using Gemini's DVFS scheme. In addition to that, we can extend the two-step DVFS for single core to a multiple-step design for improved energy savings.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283. USENIX Association, 2016.

[2] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. Energy proportional datacenter networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 338–347. Association for Computing Machinery, 2010.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74. Association for Computing Machinery, 2008.

[4] 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)*, pages 25–36, 2017.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 503–514. Association for Computing Machinery, 2014.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74. Association for Computing Machinery, 2010.

[7] Robin Aly, Djoerd Hiemstra, and Thomas Demeester. Taily: Shard selection using the tail of score distributions. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, page 673–682. Association for Computing Machinery, 2013.

[8] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):760–768, 2000.

[9] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, page 372–379. Association for Computing Machinery, 2006.

[10] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, 2009.

[11] Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2(4), October 2008.

[12] Xiao Bai, Ioannis Arapakis, B. Barla Cambazoglu, and Ana Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Trans. Inf. Syst.*, 36(2), August 2017.

[13] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 242–253. Association for Computing Machinery, 2011.

[14] 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*, IMC '10, page 267–280. Association for Computing Machinery, 2010.

[15] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, page 426–434. Association for Computing Machinery, 2003.

[16] B Barla Cambazoglu, Vassilis Plachouras, and Ricardo Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 411–418. Association for Computing Machinery, 2009.

[17] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned

ranking systems. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, page 411–420. Association for Computing Machinery, 2010.

[18] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 1–12. Association for Computing Machinery, 2007.

[19] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 390–399, 2011.

[20] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.

[21] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.

[22] C. Chou, L. N. Bhuyan, and S. Ren. Tailcut: Power reduction under quality and latency constraints in distributed search systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1465–1475, 2017.

[23] C. Chou, L. N. Bhuyan, and D. Wong. μdpm: Dynamic power management for the microsecond era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 120–132, 2019.

[24] 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, page 212–217. Association for Computing Machinery, 2016.

[25] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 407–424. USENIX Association, 2016.

[26] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, page 273–286. USENIX Association, 2005.

[27] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads

for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167. Association for Computing Machinery, 2017.

[28] Checkpoint/restore in userspace. `https://www.criu.org/Main_Page`.

[29] Constraint-based lsp setup using ldp. `https://tools.ietf.org/html/rfc3212`.

[30] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 161–174. USENIX Association, 2008.

[31] Andrew R. Curtis, S. Keshav, and Alejandro Lopez-Ortiz. Legup: Using heterogeneity to reduce the cost of data center network upgrades. In *Proceedings of the 6th International COnference*, Co-NEXT '10. Association for Computing Machinery, 2010.

[32] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, 6(11):1150–1161, August 2013.

[33] Zhuyun Dai, Chenyan Xiong, and Jamie Callan. Query-biased partitioning for selective search. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, page 1119–1128. Association for Computing Machinery, 2016.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[35] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 497–509. Association for Computing Machinery, 2016.

[36] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, page 993–1002. Association for Computing Machinery, 2011.

[37] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive. A flexible model for resource management in virtual private networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 95–108. Association for Computing Machinery, 1999.

[38] Ronald Fagin. Combining fuzzy information: An overview. *SIGMOD Rec.*, 31(2):109–118, June 2002.

[39] Introducing data center fabric, the next-generation facebook data center network. https://code.facebook.com/posts/360346274145943/.

[40] 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, page 37–48. Association for Computing Machinery, 2012.

[41] Sally Floyd. Tcp and explicit congestion notification. *SIGCOMM Comput. Commun. Rev.*, 24(5):8–23, October 1994.

[42] Gnu linear programming kit. https://www.gnu.org/software/glpk/.

[43] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62. Association for Computing Machinery, 2009.

[44] Tian Guo, Vijay Gopalakrishnan, K. K. Ramakrishnan, Prashant Shenoy, Arun Venkataramani, and Seungjoon Lee. Vmshadow: Optimizing the performance of latency-sensitive virtual desktops in distributed clouds. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, page 103–114. Association for Computing Machinery, 2014.

[45] Maruti Gupta and Suresh Singh. Greening of the internet. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 19–26. Association for Computing Machinery, 2003.

[46] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 243–254. Association for Computing Machinery, 2010.

[47] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 161–175. Association for Computing Machinery, 2015.

[48] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 625–638. Association for Computing Machinery, 2017.

[49] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 249–264. USENIX Association, 2010.

[50] Hpe altoline 6940 switch series. `https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04741125`.

[51] Hp switch software openflow v1.3 administrator guide. `https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04777809&docLocale=en_US`.

[52] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 15–28. USENIX Association, 2015.

[53] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 219–230. Association for Computing Machinery, 2013.

[54] 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*, EuroSys '13, page 155–168. Association for Computing Machinery, 2013.

[55] 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. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, page 253–262. Association for Computing Machinery, 2014.

[56] Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, and Geri Gay. Accurately interpreting clickthrough data as implicit feedback. *SIGIR Forum*, 51(1):4–11, August 2017.

[57] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, page 253–264. Association for Computing Machinery, 2005.

[58] Evangelos Kanoulas, Keshi Dai, Virgil Pavlu, and Javed A. Aslam. Score distribution models: Assumptions, intuition, and robustness to score manipulation. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, page 242–249. Association for Computing Machinery, 2010.

[59] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant os updates via userspace checkpoint-and-restart. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 605–619. USENIX Association, 2016.

[60] 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*, MICRO-48, page 598–610. Association for Computing Machinery, 2015.

[61] 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. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, page 7–16. Association for Computing Machinery, 2015.

[62] Yubin Kim, Jamie Callan, J. Shane Culpepper, and Alistair Moffat. Load-balancing in distributed selective search. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, page 905–908. Association for Computing Machinery, 2016.

[63] Diederick P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[64] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[65] Anagha Kulkarni, Almer S. Tigelaar, Djoerd Hiemstra, and Jamie Callan. Shard ranking and cutoff estimation for topically partitioned collections. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, page 555–564. Association for Computing Machinery, 2012.

[66] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07. USENIX Association, 2007.

[67] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *2011 Proceedings IEEE INFOCOM*, pages 1098–1106, 2011.

[68] Yanpei Liu, Stark C. Draper, and Nam Sung Kim. Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, page 313–324. IEEE Press, 2014.

[69] 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 Architecuture*, ISCA '14, page 301–312. IEEE Press, 2014.

[70] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with pace. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '01, page 50–61. Association for Computing Machinery, 2001.

[71] Craig Macdonald, Iadh Ounis, and Nicola Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Syst.*, 29(4), December 2011.

[72] Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. Learning to predict response times for online query scheduling. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, page 621–630. Association for Computing Machinery, 2012.

[73] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[74] 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, page 205–216. Association for Computing Machinery, 2009.

[75] 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, page 313–324. Association for Computing Machinery, 2012.

[76] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, page 1154–1162. IEEE Press, 2010.

[77] Metis - serial graph partitioning and fill-reducing matrix ordering. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[78] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550. Association for Computing Machinery, 2015.

[79] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550. Association for Computing Machinery, 2015.

[80] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, June 2007.

[81] Hafeezul Rahman Mohammad, Keyang Xu, Jamie Callan, and J. Shane Culpepper. Dynamic shard cutoff prediction for selective search. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '18, page 85–94. Association for Computing Machinery, 2018.

[82] Multiprotocol label switching architecture. `https://www.ietf.org/rfc/rfc3031.txt`.

[83] American national standard for myrinet-on-vme protocol specification. `https://cms-docdb.cern.ch/cgi-bin/PublicDocDB/RetrieveFile?docid=2705&version=1&filename=Av26.pdf`.

[84] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 323–336. USENIX Association, 2008.

[85] Open compute project server design specification. `http://www.opencompute.org/wiki/Server/SpecsAndDesigns`.

[86] Open compute project switch design specification. `http://www.opencompute.org/wiki/Networking/SpecsAndDesigns`.

[87] Open network operating system. `https://onosproject.org/`.

[88] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. *IEEE Trans. Parallel Distrib. Syst.*, 8(8):884–896, August 1997.

[89] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 117–130. USENIX Association, 2015.

[90] Pox controller. `https://github.com/noxrepo/pox`.

[91] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 342–355. Association for Computing Machinery, 2015.

[92] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Trans. Comput. Syst.*, 8(2):158–181, May 1990.

[93] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137. Association for Computing Machinery, 2015.

[94] Rsvp-te: Extensions to rsvp for lsp tunnels. `https://tools.ietf.org/html/rfc3209`.

[95] Sampled flow. `https://sflow.org/`.

[96] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. Incentivizing self-capping to increase cloud utilization. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 52–65. Association for Computing Machinery, 2017.

[97] Luo Si and Jamie Callan. Relevant document distribution estimation method for resource selection. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, page 298–305. Association for Computing Machinery, 2003.

[98] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197. Association for Computing Machinery, 2015.

[99] R. Singh, P. Shenoy, K. K. Ramakrishnan, R. Kelkar, and H. Vin. etransform: Transforming enterprise data centers by automated consolidation. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 1–11, 2012.

[100] A simple network management protocol (snmp). `https://tools.ietf.org/html/rfc1157`.

[101] Apache solr. `https://lucene.apache.org/solr/`.

[102] W. Song, Z. Xiao, Q. Chen, and H. Luo. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers*, 63(11):2647–2660, 2014.

[103] Specpower_ssj 2008. `https://www.spec.org/power_ssj2008/`.

[104] Paul Thomas and Milad Shokouhi. Sushi: Scoring scaled samples for server selection. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 419–426. Association for Computing Machinery, 2009.

[105] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, page 63–72. Association for Computing Machinery, 2013.

[106] United states data center energy usage report, 2016. `https://eta.lbl.gov/publications/united-states-data-center-energy`.

[107] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Comput. Netw.*, 53(11):1830–1845, July 2009.

[108] 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, page 585–597. Association for Computing Machinery, 2015.

[109] Nedeljko Vasić, Prateek Bhurat, Dejan Novaković, Marco Canini, Satyam Shekhar, and Dejan Kostić. Identifying and using energy-critical paths. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11. Association for Computing Machinery, 2011.

[110] Nedeljko Vasić and Dejan Kostić. Energy-aware traffic engineering. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, page 169–178. Association for Computing Machinery, 2010.

[111] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15. Association for Computing Machinery, 2015.

[112] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 243–264. Springer-Verlag New York, Inc., 2008.

[113] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao. Carpo: Correlation-aware power optimization in data center networks. In *2012 Proceedings IEEE INFOCOM*, pages 1125–1133, 2012.

[114] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 99–112. USENIX Association, 2011.

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

[116] Daniel Wong. Peak efficiency aware scheduling for highly energy proportional servers. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 481–492. IEEE Press, 2016.

[117] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus van der Merwe. Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, page 121–132. Association for Computing Machinery, 2011.

[118] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 248–259. Association for Computing Machinery, 2004.

[119] Xapian. `https://xapian.org/`.

[120] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 199–210. Association for Computing Machinery, 2012.

[121] Mingwei Xu, Yunfei Shang, Dan Li, and Xin Wang. Greening data center networks with throughput-guaranteed power-aware routing. *Comput. Netw.*, 57(15):2880–2899, October 2013.

[122] Qing Yi and Suresh Singh. Minimizing energy consumption of fattree data center networks. *SIGMETRICS Perform. Eval. Rev.*, 42(3):67–72, December 2014.

[123] Jeong-Min Yun, Yuxiong He, Sameh Elnikety, and Shaolei Ren. Optimal aggregation policy for reducing tail latency of web search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, page 63–72. Association for Computing Machinery, 2015.

[124] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16. Association for Computing Machinery, 2016.

[125] M. Zhang, C. Yi, B. Liu, and B. Zhang. Greente: Power-aware traffic engineering. In *The 18th IEEE International Conference on Network Protocols*, pages 21–30, 2010.

[126] K. Zheng and X. Wang. Dynamic control of flow completion time for power efficiency of data center networks. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 340–350, 2017.

[127] K. Zheng, X. Wang, L. Li, and X. Wang. Joint power optimization of data center network and servers with correlation analysis. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 2598–2606, 2014.

[128] K. Zheng, X. Wang, and J. Liu. Disco: Distributed traffic flow consolidation for power efficient data center network. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9, 2017.

[129] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan. Goldilocks: Adaptive resource provisioning in containerized data centers. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 666–677, 2019.

[130] L. Zhou, C. Chou, L. N. Bhuyan, K. K. Ramakrishnan, and D. Wong. Joint server and network energy saving in data centers for latency-sensitive applications. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 700–709, 2018.

[131] Liang Zhou, Laxmi N. Bhuyan, and K. K. Ramakrishnan. Dream: Distributed energy-aware traffic management for data center networks. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, e-Energy '19, page 273–284. Association for Computing Machinery, 2019.

[132] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, December 1998.