DeepPower: Deep Reinforcement Learning based Power Management for Latency Critical Applications in Multi-core Systems

Jingrun Zhang, Guangba Yu, Zilong He, Liang Ai, Pengfei Chen*
Sun Yat-sen University
China
{zhangjr35,yugb5,hezlong,ailiang3}@mail2.sysu.edu.cn,chenpf7@mail.sysu.edu.cn

ABSTRACT

Latency-critical (LC) applications are widely deployed in modern datacenters. Effective power management for LC applications can yield significant cost savings. However, it poses a significant challenge in maintaining the desired Service Level Aggrement (SLA) levels. Prior researches have mainly emphasized predicting the service time of request and utilize heuristic algorithms for CPU frequency adjustment. Unfortunately, the control granularity is limited to the request level and manual feature selection is needed.

This paper proposes DeepPower, a deep reinforcement learning (DRL) based power management solution for LC applications. DeepPower comprises two key components, a DRL agent for monitoring the system load changes and a thread controller for CPU frequency adjustment. Considering the high overhead of the neural network and the short service time of requests, it is infeasible to employ DRL for direct adjustment of CPU frequency at the request level. Instead, DeepPower proposes a hierarchical control mechanism. That means the DRL agent adjusts the parameter of thread controller with longer intervals, and thread controller adjusts the CPU frequency with shorter intervals. This control mechanism enables DeepPower to adapt to dynamic workloads and achieves fine-grained frequency adjustments. We evaluate DeepPower with some common LC applications under dynamic workload. The experimental results show that DeepPower saves up to 28.4% power compared with state-of-the-art methods and reduces the percentage of request timeout.

CCS CONCEPTS

• **Computer systems organization → Client-server architectures**.

KEYWORDS

latency-critical system, power management, deep reinforcement learning, hierarchical control

1 INTRODUCTION

Latency-critical applications occupy a high proportion in modern datacenters, providing users with various services [14] Datacenter operators need to ensure that the Quality-of-Service (QoS) constraint of these applications is within an agreed level. Generally, the tail latency of a service cannot exceed a specified threshold, typically on the order of milliseconds, to ensure an acceptable level of user experience [23, 26]. However, the variation in request workload over time and the variation in the execution time of each request is non-negligible in latency-critical applications. Datacenters typically maintain high CPU frequencies to meet the latency constraints, resulting in low utilization and excessive power consumption [2, 8, 24]. Therefore, QoS-aware fine-grained power management can deliver substantial economic benefits for companies managing large-scale datacenters[2, 12, 21, 27].

The good news is modern processor vendors are providing more and more power optimization techniques. Dynamic voltage/frequency scaling (DVFS)[15] technology makes it possible to dynamically adjust the frequency of each thread at runtime with a delay in a few microseconds. Therefore, a straightforward idea is to set the CPU frequency at a high level to improve its computing capability under a high load, thus ensuring the QoS requirements. When the load is low, the CPU frequency is scaled down to reduce power consumption. However, the workload of latency-critical applications changes frequently, and the service time of requests follows a long-tail distribution, increasing difficulty for the trade-off between QoS and power consumption

Many efforts have been devoted in this field, focusing on the trade-off of CPU frequency and response time [5, 10, 12, 19, 31]. Adrenaline [10] divides the requests into long requests and short requests based on expert knowledge, and boosts the frequency for long requests. Rubik [12] models the distribution of requests' service time, and proposes a heuristic method for scaling frequency for each request based on statistical analysis. Gemini [31] leverages neural networks to predict the service time of a request for web search based on features, and proposes a two-stage frequency selection method. Retail [5] demonstrates that a simple linear regression is accurate enough for service time prediction and uses a heuristic method to determine the frequency of each request.

The common point of these approaches is that they model the latency of requests assuming the request workload is static (i.e. request-per-second (RPS) does not change over time). We conduct experiments to point out that the model accuracy deteriorates under a dynamic workload. A heuristic algorithm for CPU frequency scaling is simple, and a more advanced policy could lead to a more significant power reduction. Moreover, the previous methods scale the frequency at the granularity of requests. However, the experiment results show that a more fine-grained method results in better performance.

In this paper, we propose DeepPower, a Deep Reinforcement Learning (DRL) based hierarchical control power management framework for latency-critical applications. DRL is known for the capability to learn complex policies and adaptability to dynamic and complex scenarios, but has been criticized for the overhead of

Table 1: Comparison of DeepPower and other methods

	Feature Engineering	Expert Knowledge	Control Level	Dynamic Workload Adaptability	Overhead	Method
Rubik [12]	No	No	Request level	Medium	Low	Statistical model
Adrenaline [10]	Yes	Yes	Request level	Medium	Low	Expert rules
Gemini [31]	Yes	No	Request level	Bad	High	Neural network
Retail [5]	Yes	No	Request level	Medium	Low	Linear regression
DeepPower	**No**	**No**	**Time slice level**	**Good**	**Low**	**DRL-based hierarchical control**

neural networks [3]. Since the inference time of a simple neural network can take a hundred microseconds, it is challenging to utilize DRL for a fine-grained level (e.g., user request level) control. We overcome this challenge by proposing a hierarchical control mechanism. The top layer outputs an action in a longer interval, and trains the neural network based on the state transition and reward function. Meanwhile, the bottom layer selects a frequency for each CPU core in shorter intervals, guided by the action of the top layer. The mechanism that separates policy learning from frequency selection enables our method to achieve both fine-grained control and the ability to learn complex policies.

With the powerful adaptability and representation abilities of DRL, DeepPower is robust to dynamic loads and can learn an optimal policy. Meanwhile, DeepPower does not depend on specific features and expert knowledge, leading to improved generalization. Moreover, with the innovative hierarchical control mechanism, DeepPower saves more energy than other methods while meeting the QoS requirements. The differences between DeepPower and the state-of-the-art methods are listed in Table 1, from six dimensions, namely whether need feature engineering, whether dependent on expert knowledge, the control level, the dynamic workload apatability, the introduced overhead, and the adopted method.

Compared with state-of-the-art methods such as Gemini [31] and ReTail [5], DeepPower can reduce power consumption by up to 28.4% while satisfying tail latency constraints. Moreover, we conduct extended experiments to demonstrate that DeepPower can be generalized to different scenarios, introducing a negligible overhead.

In summary, this paper makes the following contributions:

- We experimentally prove that the request time prediction based methods are not accurate enough when the workload changes dynamically. DeepPower does not need manual feature engineering for prediction, which increases its generization.
- As far as we know, DeepPower is the first method to apply DRL for power management in latency-critical applications. Due to the powerful learning capability of DRL, the learned policy is more efficient than heuristic methods, resulting in more power reduction.
- By using a hierarchical control mechanism, we push CPU frequency scaling to a more fine-grained level, leading to more energy savings while preventing requests from timing out.

2 BACKGROUND

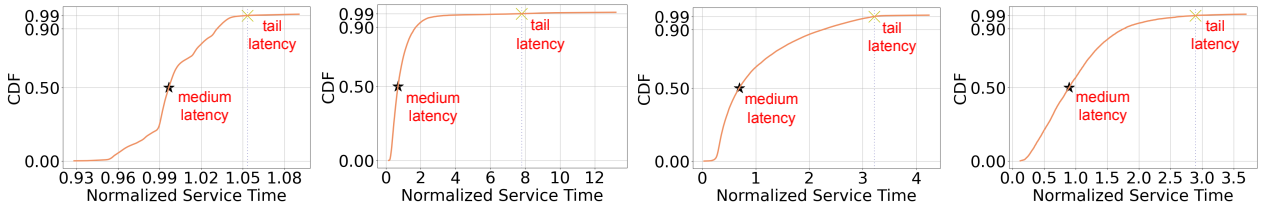
2.1 Latency-Critical Application

A large number of applications are deployed across thousands of computing nodes in datacenters to provide services to users. A request sent from clients to the datacenter will be processed sequentially. In order to make sure that users receive responses in a short period, the tail latency of each request should be maintained below a specific level, usually a few milliseconds. In this work, we solely focus on the power management of latency-sensitive applications within a single computing node, such as virtual machine or physical machine. Generally, multiple threads run on one node, with each thread scheduled to a physical CPU core. DVFS can be used to scale the frequency of the core without affecting the performance of other cores.

The variability of request processing time poses a significant difficulty in power management. We demonstrate this variability in service time using data collected from four latency-critical applications from Tailbench [14]. Fig. 1 shows the Cumulative Distribution Function (CDF) of normalized service time. We can observe that a long-tailed distribution exists in the CDF. For example, in the Moses application, tail latency is approximately 8 times larger than the average service time. Since a small percentage of requests' processing time significantly exceeds the average time, they make an intractable obstacle for ensuring Quality-of-Service.

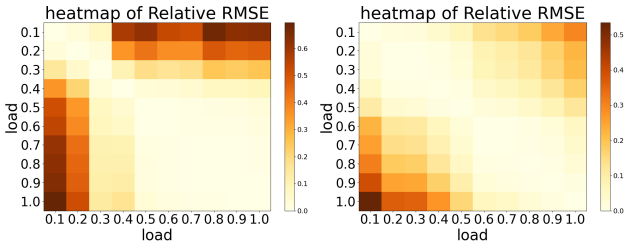
2.2 Service Time Modeling

Given that requests with longer service time are the major challenge, previous works have proposed extensive solutions. The common pattern of these solutions is predicting the service time of requests and setting a higher frequency for requests with long service time. Retail and Gemini are exemplary methods in this field, utilizing neural networks and linear regression, respectively, to predict request execution times. Gemini [31] created a two-stage frequency boost method utilizing the prediction model. The method sets a baseline frequency, and will increase it to the maximum frequency if the queue of waiting requests risks timing out. Retail [5] selects the minimum frequency at which the execution of all requests in the queue will not result in a timeout. Then Retail uses this frequency to execute the first request in the queue. Since the prediction-based approach has a accurate prediction capability under static loads, Gemini and Retail obtain a certain power-saving effect.



(a) Img-dnn (b) Moses (c) Xapian (d) Sphinx

Figure 1: CDF of service time divided by the mean for four latency-critical applications.



(a) Masstree (b) Sphinx

Figure 2: Heatmap of Relative Root Mean Square Error(RMSE). The value of row i and column j in the heatmap means the relative RMSE generated by using the model from $level-i$ loads to predict the service time of $level-j$ in Masstree and Sphinx applications.

3 MOTIVATION

3.1 Load-Aware Power Management

Gemini and Retail have made significant progress by predicting the service time of requests. However, the prediction may become inaccurate when the load changes. Since many threads process the requests in the same machine, different threads have contention for memory, cache, and disk. Although the application colocation has been studied [4], it is still complex and intractable to quantify this contention. When the RPS changes, the impact of this contention on service time also varies together, which may mislead the prediction.

We demonstrate it with a simple experiment with Masstree and Sphinx applications. Linear regression models described in [5] are adopted to train with data collected from different load levels, denoted as $model_i = train(data_i)$. Then we use the service time model from one load to predict data from other loads and get $error(i, j)$ as Root Mean Square Error (RMSE) of $predict(model_i, data_j)$. Finally, define Relative RMSE (i, j) as $error(i, j)/error(j, j)$, i.e., the prediction error after the load changes. Fig. 2 shows the heatmap of Relative RMSE. However, when the load changes substantially, the prediction becomes inaccurate. To this end, the power management method should be workload-aware.

3.2 DRL-based Hierarchical Control

DRL utilizes a deep neural network for value function or policy approximation. However, the overhead of the neural network is too high to request-level frequency scaling. We demonstrate it with a simple experiment. Four classic DRL algorithms are implemented with a lightweight neural network. The client sends request information to the server, and the DRL agent generates an action. Then the inference time of each algorithm is recorded.

Table 2: Inference time of each DRL algorithm.

	DQN[22]	DDQN[28]	DDPG[17]	SAC[9]
Inference time (us)	125	140	231	472

The inference time of each algorithm is shown in Table 2. The value may not appear to be significant, but the service time of requests in many latency-critical systems is sub-millisecond, and RPS may reach a million with multi-threaded [14]. Therefore, if DRL is applied for request-level frequency scaling, the heavy overload will severely hurt the performance. Although some researchers have applied reinforcement learning to power management [16, 29, 30, 32], as far as we know, DeepPower is the first DRL-based power management method for the latency-critical applications.

To tackle this challenge, we propose a DRL-based hierarchical control mechanism. The top-level DRL agent considers the dynamic changes in workload but avoids directly adjusting the CPU core frequency. Instead, it generates the parameters for the bottom-level algorithm that handles the fine-grained frequency scaling.

4 DEEPPOWER FRAMEWORK

4.1 Overview

The framework of DeepPower and its interaction with a latency-critical system is illustrated in Fig. 3. In a latency-critical system, requests from the user client are sent to the server's queue. The server runs many working threads, which fetch requests from the queue and process them without preemption. Each thread is combined with a CPU core, while the thread controller scales the core frequency with the DRL Agent's guideline. The server collects comprehensive information about the system (i.e. the number of timeout requests, the length of queue) and sends it to DeepPower framework.

The DeepPower framework mainly consists of two components: the DRL agent and the thread controller, and also includes a state observer, a reward calculator, and a power monitor. The state observer receives information about a long time past of the latency-critical system and produces a normalized state vector which will be utilized by the policy neural network (actor) of the DRL Agent. The actor generates an action based on the state, and the action is used as parameters of the thread controller. The thread controller continually scales the frequency of all CPU cores according to the action and the remaining time of the request. The reward calculator relies

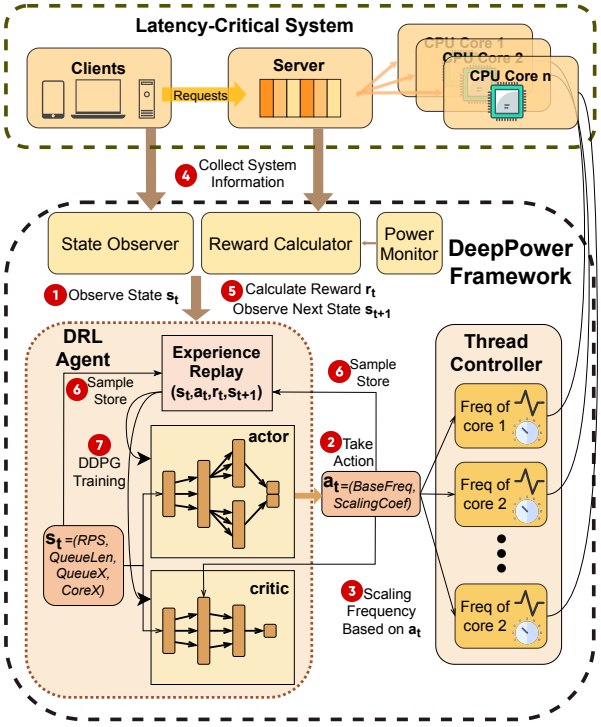


Figure 3: Overview of DeepPower framework for latency-critical system power management.

on both the information from the latency-critical system and the energy consumed from the power monitor to calculate rewards. State transitions are collected in the experience replay pool, from which the DRL agent samples a batch of transitions and trains the neural network. As aforementioned, the DRL agent takes actions at longer intervals, while the thread controller scales frequency at shorter intervals.

4.2 Thread Controller

As depicted in Fig. 1, the distribution of requests service time is highly-skewed, i.e., a few requests consumed a longer processing time. Prior methods identify these long requests and boost the CPU frequency for them. Differently, we gradually increase the frequency according to the request processing time. Intuitively, when a request takes a long time to process, it should be sped up.

Thread controller (Algorithm 1) receives two non-negative parameters from DRL agent, *BaseFreq* and *ScalingCoef*, and calculate a score based on the parameters and the begin time of request. CPU core will be set to turbo if score greater than 1, means the request processing needs to be completed as soon as possible. Otherwise, a linear interpolation betweem the max and min frequency will be done based on the score, and get the frequency of CPU core. Since Scaling frequency operation can be finished in a few microseconds, thread controller can scale the frequencies of all cores in less than a millisecond.

Fig. 4 depicts the change in frequency of a CPU core over a period of 2000 milliseconds, during which the frequency level is recorded per millisecond. If there is no request processing, the frequency

Algorithm 1: Thread Controller

Input:
$BeginTimes$: Request arrive time of each thread
$BaseFreq, ScalingCoef$: Base frequency and the degree to which frequency varies with consumed time
$Freq_{min}, Freq_{max}$: Min and Max frequency of system
SLA: Latency requirement of Requests

while *True* **do**
  $curTime = getTime()$
  for $i = 1$ *to* $ThreadNum$ **do**
    $consumedTime = \frac{curTime - beginTimes[i]}{SLA}$
    $Score = consumedTime * ScalingCoef + BaseFreq$
    if $Score >= 1$ **then**
      Scale the frequency of $i-th$ core to Turbo
    else
      $freq = Freq_{min} + (Freq_{max} - Freq_{min}) * Score$
      Scale the frequency of $i-th$ core to $freq$
    end
  end
  $sleep(ShortTime)$;
end

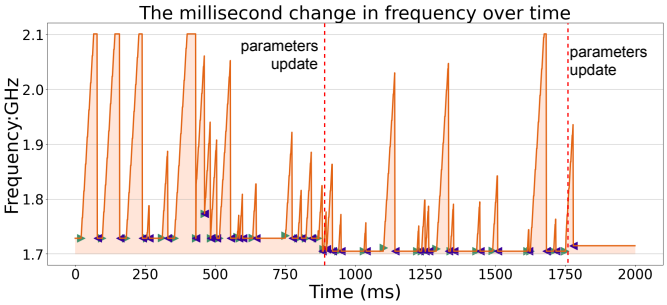


Figure 4: Millisecond-level frequency in the control of thread controller during 2 seconds. The time of requests processing begin and end are marked with green and blue marker. The time when the parameter is updated is marked with red dot line.

is set to *BaseFreq*; otherwise, it is gradually scaled up. The scaling is determined by the value of *ScalingCoef*. In contrast, shorter requests are finished quickly even with a lower frequency, while for longer requests, a higher frequency is used to prevent timeouts. The *BaseFreq* and *ScalingCoef* parameters of the thread controller are updated by the DRL agent during the 2-second period. Thanks to this fine-grained and flexible control mechanism, DeepPower achieves better power savings and prevents SLA violations.

4.3 DRL Problem Formulation

Our method is appropriate for typical latency-critical application scenarios. A server is located exclusively on one machine with a multi-core processor. The processor can adjust frequencies with a microsecond's delay and offers various frequency options ranging from hundreds of megahertz to several gigahertz, including support for turbo boost.

Requests are sent to a server queue for processing, where several threads retrieve and process them. Latency is defined as the time between when a request arrives at the server and when it is sent back to the client. Quality-of-Service (QoS) refers to tail latency, which is the latency value at the 99th percentile. QoS should be guaranteed to be less than a specified value (also referred as SLA), typically measured in milliseconds.

Previous studies presume that the inter-arrival times of requests conform to an exponential distribution whose parameter remains constant over time, resulting in a stable RPS. However, this does not hold true in the actual scenario. In reality, RPS exhibits a diurnal pattern, as previously demonstrated in [19, 20, 24], such as requests in the afternoon are generally more than in the early morning. To simulate this variation, we use a realistic workload that entails periodic and significant variations in the number of requests sent by the client.

We denote the power consumption at time t as $P(t)$, the number of timeout requests at time t as $TR(t)$ and the RPS as $RPS(t)$. Then the objective which is minimizing the total power consumpption while satisfying QoS constraint over T timestep can be formulated as an optimization problem as follows:

$$\min_{\pi} \sum_{t=1}^{T} P(t) \tag{1}$$

$$s.t. \sum_{t=1}^{T} TR(t) \leq \sum_{t=1}^{T} RPS(t) * 0.01 \tag{2}$$

In Eq. 1, π denotes the policy learnt by the DRL agent. The policy determines the parameters of thread controller $BaseFreq_t, ScalingCoef_t$ at each time t. Lower values of the parameters aid in reducing power consumption; however, this comes at the trade-off of an increased risk of request timeouts. The QoS constraint is converted as the percentage of timeout requests less than 0.01 in Eq. 2. The problem can be transformed into a $MDP =< S, A, R, \rho, \gamma >$ which will be elaborated in the following. Considering that the parameter is a continuous value, we employ the deep deterministic policy gradient algorithm (DDPG). The agent aims to maximize the expectation of policy π_θ,

$$\begin{aligned} J(\pi_\theta) &= \int_S \rho^\pi(s) \int_A \pi_\theta(s,a) r(s,a) da ds \\ &= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[r(s,a)] \end{aligned} \tag{3}$$

where θ denotes the parameters of policy π, ρ^π denotes the state distribution under policy π and $\mathbb{E}_{s\sim\rho}[\cdot]$ denotes the expectation under the state distribution.

4.4 DRL Agent

The DRL Agent catches the dynamic changes in the system workload and generates an action for the thread controller. We detail the design of the state space and reward function of the agent and the principles behind them.

4.4.1 State Space. Apparently, DeepPower should increase the frequency when the workload is heavy to avoid request timeouts and decrease it for power saving if the workload gets lighter. So the state should reflect the condition of the workload. Besides, the state space should be as small as possible to reduce the overhead of training and inference and facilitate the learning of neural networks. Therefore, it is indispensable to model the workload condition with refined data accurately.

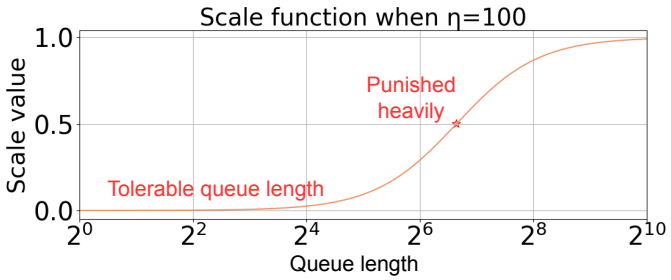


Figure 5: The scale function when $\eta = 100$. The change point is marked with a red pentagram.

An essential measure of load is obviously the RPS. Besides, the number of requests being queued is also meaningful since queue time has a significant impact on the latency [11–13]. Moreover, if many requests are about to time out, it may be necessary to increase the CPU frequency. Based on the above analysis, DeepPower uses an 8-dimensional vector (*NumReq*, *QueueLen*, *Queue25*, *Queue50*, *Queue75*, *Core25*, *Core50*, *Core75*) to represent the workload condition of the system:

- NumReq: The number of Requests received in the last period.
- QueueLen: The length of the queue in the server, which means how many requests are in the queue.
- QueueX: The number of requests in the queue whose time remains less than SLA*X%.
- CoreX: The number of requests which are being processed and their time remaining less than SLA*X%.

4.4.2 Reward Function. The reward function is crucial for the reinforcement learning task, and the influence of many aspects on the system must be taken into account. We design the reward function with consideration of three aspects, energy consumed, timeout requests, and the length of the queue, and linearly combine them to the total reward function:

$$R_{total} = -(\alpha \times R_{energy} + \beta \times R_{timeout} + \gamma * R_{queue})$$

- R_{energy}: Measure the power consumption in the previous time step, which is closely related to CPU frequency.
- $R_{timeout}$: Measure how many requests timeout in last time step.
- R_{queue}: If too many requests are in the queue, significant queuing latencies will incur. Punishing the agent when the queue is longer than last time step is an intuitive idea. However, the situation that the queue is short and the number of requests increases does not hurt the performance. Therefore, we set R_{queue} as the multiplication of how much the queue is lengthened and a scaling function. In formal terms, we denote ql_t as the length of queue at time t, and the define R_{queue} at time t as:

$$R_{queue} = scaleFunc(ql_t) * max(ql_t - ql_{t-1}, 0)$$

$$scaleFunc(x) = \frac{\frac{x}{\eta}}{\frac{x}{\eta} + \frac{\eta}{x+\epsilon}}$$

$scaleFunc(x)$ is substantially closed to 0 when x is less than η and converges to 1 when x goes to infinity, as shown in Fig. 5. The hyper-parameter η determines the threshold when the queue becomes longer, i.e. , the agent won't get severely punished if the length of queue is below η, while a large negative reward will be received when the length of queue is above η and the queue becomes longer. With the help of $scaleFunc$ and hyper-parameter η, the agent is instructed to prevent queues from becoming too long.

Linear combination makes the reward function contain three aspects of information. Changing the weight of each term leads to adjusting the DRL Agent's training objectives. For example, we can increase the value of β to improve the importance of $R_{timeout}$ if we find that the tail latency is higher than the SLA metric.

4.4.3 Action Space. The action of DRL agent is two parameters of the thread controller, $BaseFreq, ScalingCoef$. We use a simple neural network, consisting of fully connected layers and a sigmoid function, to limit the parameter values to the range of $[0, 1]$.

4.5 Training Process

Algorithm 2: DRL agent training process

Input: Four neural network π_θ,$\pi_{\theta'}$,Q_w,$Q_{w'}$, Replay pool D, Latency-Critical System LC, State Observer SO, Reward Calculator RC, Thread Controller TC, Power Monitor PM

Output: Actor π_θ

```
Run LC, initialize SO, RC, TC, PM
Get initial state from SO as s_1
for t = 1 → T do
    if t ≥ WARMUP then
        a_t=π_θ(s_t) + N(μ, δ)
    else
        a_t=randomSelect()
    end
    SEND a_t to TC,sleep(LongTime)
    SO and RC get information from LC,PM
    Draw s_{t+1} from SO and r_t from RC
    Push transition s_t, a_t, r_t, s_{t+1} in D
    if t ≥ WARMUP then
        Sample a batch transition (s_i, a_i, r_i, s'_i) from D
        Target Q value y_i = r_i + γQ_{w'}(s'_i, π_{θ'}(s'_i))
        L_c = Σ_i (y_i − Q_w(s_i, a_i))^2
        L_a = Σ_i −Q_w(s_i, π_θ(s_i))
        Update π_θ,Q_w with gradient descent, Soft update
          π_{θ'},Q_{w'}
    end
end
```

We exploit DDPG algorithm to learn a deterministic policy from state to action. DDPG utilized four neural network, actor π_θ works as policy network which takes state as input and produce action and critic Q_w judge the actions chosen by the actor. Target actor π'_θ and Q'_w are used to calculate target action and Q value, contributing to the stability of traning process. State transision are stored in a experience replay pool, from which agent draws a batch of samples and trains on them. Meanwhile, DDPG adds noise to the action generated by the actor , aims to increase exploratory.

The complete training algorithm is shown in algorithm 2. We first run the latency-critical system and the power monitor, and connect it with other components (the state observer, the reward calculator and the thread controller) in DeepPower framework. Action is selected randomly in the initial stage, and then is decided by the output of actor adds a normally distributed noise to explore more diverse strategies. When an action is generated, it will be sent to the thread controller, which scales the frequency of each CPU cores according to the guideline of the action. After the action has been executed for some time, the latency-critical system and the power monitor feedback information about requests and power consumption. The state observer and the reward calculator convert them into next state and reward of RL paradigm. Such a state transition is pushed into experience replay pool and will be used for training after warmup. The calculation of target Q value and loss function is identical with classic DDPG algorithm, refer to [17] for more detail.

4.6 Implementation Detail

We implement our DRL Agent with popular neural network framework Pytorch. The actor is designed as full-connected neural network with three hidden layers which has 32,24,16 neuron units seperately and ReLU is chosed as activation function. The input state passes the first shared full-connected layer and the gets throught two seperate full-connected layer, as shown in Fig. 3. A sigmoid operation is conducted on the output to keep the final action $BaseFreq, ScalingCoef$ non-negative. As for critic, we concantence the output of first hidden layer with action, and the passes throught two full-connected layer. The DRL Agent can learn efficient policy while keep the overhead low with this light-weight neural network.

In the early stages of training, the actor network tends to generate random actions. Queue congestion may take place when heavy loads arrive. Consequently, we add a noise sampled from normal distribution $N(\mu, \delta)$ to the action generated by actor and set the μ, δ to 0.3,1 by default, respectively. A positive mean leads to high frequency and avoids queueing. Large variance helps the DRL Agent explore more diverse actions and prevents it from falling into local optimal values. The $ShortTime$ and $LongTime$ are set to 1 milliseconds and 1 seconds by default, and can be changed according to the service time of different applications.

5 EVALUATION

We first evaluate DeepPower with some typical latency-critical applications and compare it with prior methods. We conduct a thorough analysis of DeepPower, exposing its superiority over other methods. Finally, we discuss the overhead of DeepPower.

5.1 Benchmark

We use 5 latency-critical applications from Tailbench for the experiment: Xapian, an open-source search engine widely used in websites and software frameworks ; Masstree, a high-performance and scalable key-value store; Moses, a statistics-based real-time

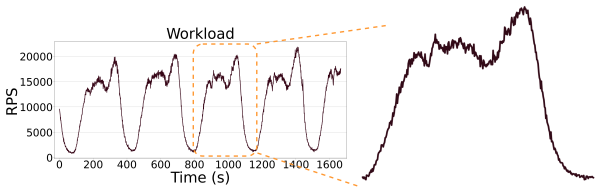


Figure 6: Changing of workload over time.

machine translation (SMT) system; Sphinx, a speech recognition system and Img-dnn, a deep neural network based image recognition program. More detailed information about those applications is summarized in Table 3.

Table 3: Benchmark information

		Xapian	Masstree	Moses	Sphinx	Img-dnn
Configuration and Dataset		English Wikipedia	mycsb-a 90% PUTs 10% GETs	Spainish articles	CMU AN4	MNIST
SLA(ms)		8	1	120	4000	5
99^{th}% ile	20%	2.742	0.191	30.99	1759.8	2.302
latency	50%	3.614	0.402	77.92	2040.7	2.295
at load	70%	4.617	0.657	100.49	2292.8	2.476

5.2 Experiment setting

The experiment is conducted on a common datacenter server. It has an Intel Xeon Gold 5218R CPU with 2 sockets and 40 cores. Besides, the server has 252G DDR4 memory and runs with Ubuntu 18.04.6. The frequency range from 0.8GHz to 2.1GHz and can be scaled with the help of the "userspace" governor of the Linux ACPI frequency driver. The energy consumption is recorded in Machine Specific Register (MSR) and can be read with Intel Running Average Power Limit (RAPL) interface [25].

The DeepPower framework is implemented with Python3 and Tailbench is implemented with C++, so a TCP connection is built for communication. The 20 worker threads [1] of Tailbench are deployed on socket0 and other threads (i.e., the threads sending/receiving requests and the thread communicating with the DRL agent) are deployed on socket1. RAPL interfaces are called to get the power consumption of socket0 during the program running.

Other works assume that the inter-arrival times of requests follow an exponential distribution and use a poisson process to simulate the workload. However, real world workload usually have a diurnal pattern [19, 20, 24] which means RPS dynamically changes over time (e.g., RPS in the afternoon obviously higher than at night). Therefore, we utilize the E-commerce search benchmark [1], which records RPS of an e-commerce search system during one month, as shown in Fig. 6. We downsample the time series to shorten the period (360s by default) and multiply the RPS by a factor to make the tail latency close to SLA when running without frequency scaling.

We train the DRL agent with a long running workload and save the neural network parameters after training. During testing, we run the DeepPower framework with a short workload and record the latency of each request. After the execution of the workload, the power, the percentage of timeout requests, the mean latency, and the tail latency will be calculated as experiment results. We also compare DeepPower with two state-of-the-art QoS-aware power management methods depicted as follows.

- **Gemini**. Gemini exploits a neural network to predict the service time of requests and uses a two-step frequency scaling strategy. While Gemini needs manually selected features, we only take features that are available when requests arrive into account. The neural network is lightweight in our experiment since the feature space is simple.
- **Retail**. Retail uses a linear regression for service time prediction and considers both request and application features. A heuristic algorithm is applied for frequency scaling when a request begins processing. Retail also monitors the tail latency of the system and adjusts the target QoS.

5.3 Experiment result

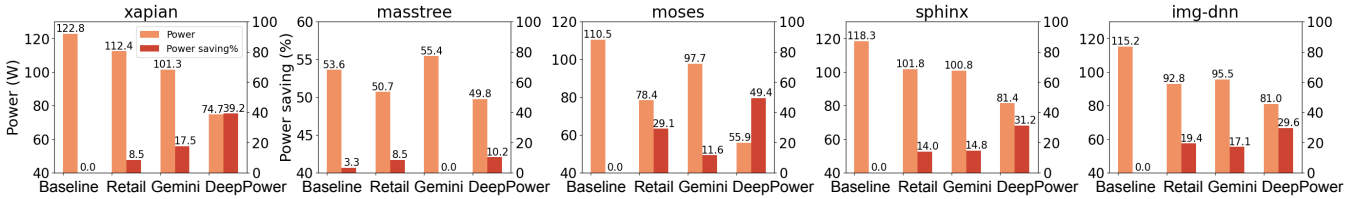
The experimental results are illustrated in Fig. 7. The performance of a power management method should be assessed through many aspects. We select the following metrics for evaluation: power consumption, power saving, mean and tail latency, and timeout rate.

We demonstrate each method's power consumption and saving effect in Fig. 7a. The baseline runs without any power management and exploits the maximum computing ability, causing tremendous power consumption. Compared to the baseline, a large amount of power saving effect has been achieved by DeepPower, except Masstree. In Sphinx and Img-dnn, DeepPower saves about 30% of energy, and this number grows to 39.7% and 49.4% for Xapian and Moses. Although Retail and Gemini are able to achieve power savings to a mild degree, DeepPower outperforms them by 12.7% (Img-dnn) to 28.4% (Moses). Note that the result on Masstree is not remarkable since only 8 threads are used, and the power consumption of the machine itself accounts for a large proportion.

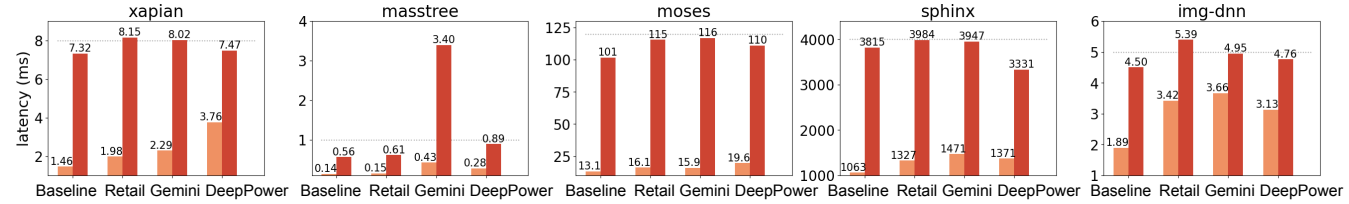
DeepPower not only reduces power consumption but also meets the QoS constraint. As shown in Fig. 7b, DeepPower keeps the tail latency lower than the SLA metric in all applications. However, Retail and Gemini slightly violate the SLA in Xapian. More seriously, the tail latency of Gemini in Masstree is more than three times SLA, which is unacceptable in practice. We ascribe it to the contradiction between the complex control mechanism of Gemini and the microsecond-level request processing time of Masstree. DeepPower guarantees better service level agreement while saves more energy, revealing its strength over other methods.

The superiority of DeepPower is also demonstrated by the mean/tail rates (i.e. mean latency divided by tail latency) and timeout rates in Fig. 7c. The mean/tail rates of DeepPower are higher than other methods in all applications except Img-dnn. DeepPower acquires a 0.5 mean/tail rate in Xapian, far above Retail and Gemini. Besides, the percentage of timeout requests of DeepPower is lower than Retail and Gemini, even if DeepPower is better at reducing power consumption. High mean/tail rates mean that DeepPower scales down the frequency for short requests to save energy, and low timeout rates mean DeepPower scales up the frequency for long requests to maintain QoS requirements.

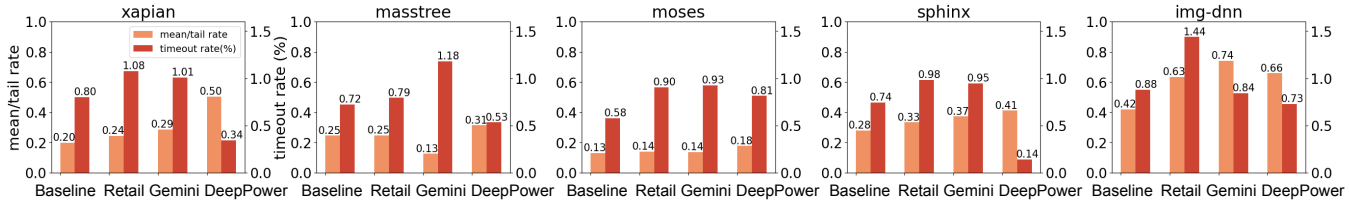
[1] 8 worker threads of Masstree since its memory overhead



(a) Power consumption and power saving for each power management method.



(b) The mean and tail latency for each power management method. SLA is denoted as grey dotted line.



(c) The mean/tail rates and requests timeout rates for each power management method.

Figure 7: Comparation between DeepPower and others state-of-the-art methods

We attribute the outstanding performance of DeepPower to two points. (i) DeepPower allows fine-grained granular frequency control. In Retail, the frequency of requests is decided when the request begins processing. Gemini steps further by boosting the frequency when the request is at risk of timeout. DeepPower scales the frequency of each thread in milliseconds or microseconds, which leads to better power saving and timeout avoidance.(ii) DeepPower is more adaptive to the dynamic workload. Previous works assume that the mean RPS is constant and adopt the same policy for different workloads. DeepPower utilizes the feedback learning mechanism of DRL, which will learn to adapt to changes in RPS with the interaction from the environment.

5.4 Deeper into DeepPower

We have demonstrated the effectiveness of DeepPower and now aim to understand the reasons behind its success furtherly. Through this analysis, we hope to shed light on the mechanisms driving the effectiveness of DeepPower.

In Fig. 8, we visualize some critical indicators over time of the running of Xapian within DeepPower. The variation curve of the power consumption basically matches the RPS, showing that DeepPower saves energy very well at low loads. DeepPower raises the *ScalingCoef* of ThreadController in high loads to avoid the queueing and timeouts and maintains *BaseFreq* at a moderate level to avoid using higher frequencies for short requests. The average frequency curve also reveals the adaptability of DeepPower. The frequency is mainly determined by *BaseFreq* when the load is low, and the influence of *ScalingCoef* grows greater when the load increases. Note that DeepPower adjusts the parameters per second to adapt to the changing workload quickly.

To further understand the advantages of DeepPower, we analyze the frequency change during the program running under each power management method. We visualize the running process of millisecond-level latency applications Xapian in Fig. 9 and second-level latency applications Sphinx in Fig. 10. DeepPower achieves fine-grained control by gradually scaling up the frequency during the request's processing, as shown in Fig. 9a and Fig. 10a. During request processing, the thread controller increases the frequency slowly. As a result, the frequency is not boosted to its maximum level most of the time. Conversely, Retail and Gemini select the frequency at a coarser granularity (i.e., once or twice per request). It makes them have to boost the CPU frequency in many cases (i.e., the occurance of long queues or request time out) and causes a large amount of energy consumption [18].

To attain a more profound understanding of the parameters *BaseFreq* and *ScalingCoef*, we execute the Xapian application with the parameters set to a fixed value and collect frequency information. Figure 11 shows the frequency changes during the different executions of Xapian with various parameter settings. Fig. 11a illustrates the result of a setting with low *BaseFreq* and high *ScalingCoef*. The low *BaseFreq* results in a lower frequency during the initial execution of requests, which is indicated on the figure by a cooler color. Meanwhile, a higher value of *ScalingCoef* causes a rapid increase of frequency during request processing. Conversely, a higher *BaseFreq* and lower *ScalingCoef* leads to a more moderate frequency changing, as Fig. 11c demonstrates. Overall, a higher *BaseFreq* is suitable for situations with heavier loads to accelerate request processing,

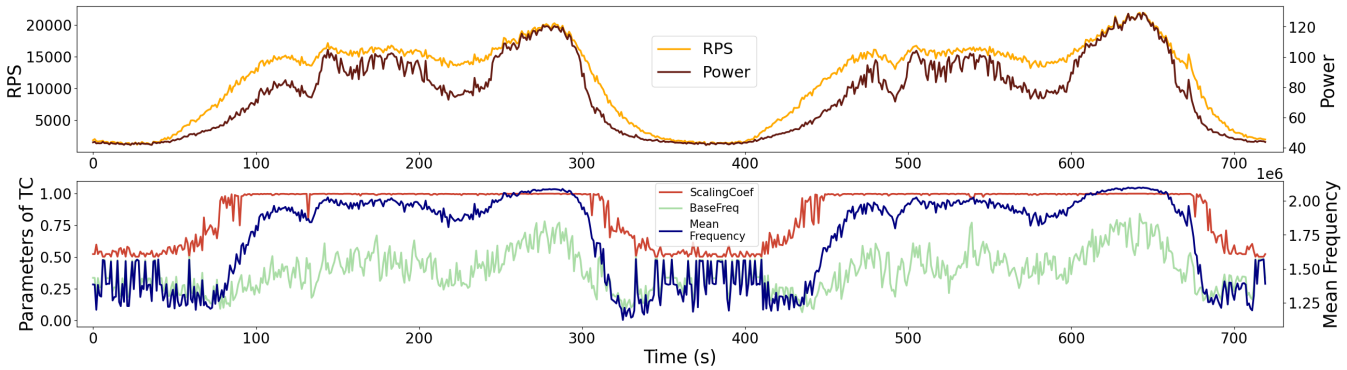
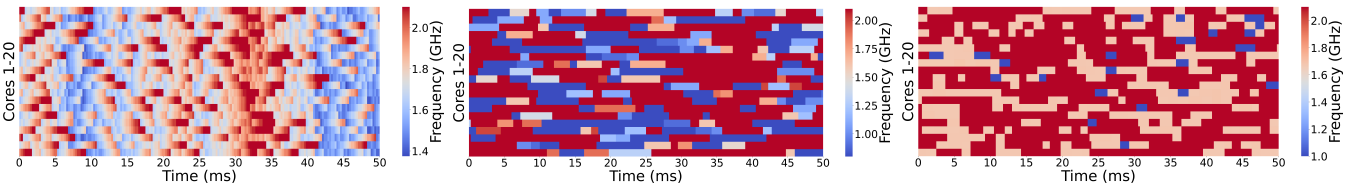
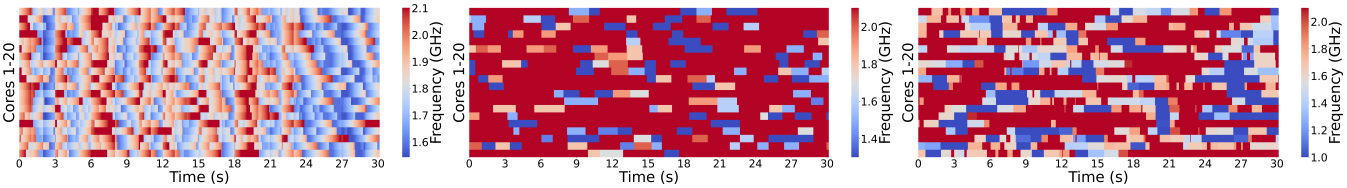


Figure 8: Visualization of RPS, power consumption, thread controller parameters over time



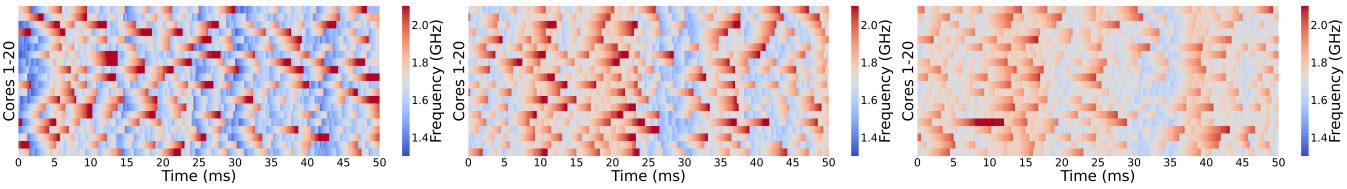
(a) DeepPower **(b) Retail** **(c) Gemini**

Figure 9: Comparison of the change in frequency of each CPU core during a 50ms execution of Xapian.



(a) DeepPower **(b) Retail** **(c) Gemini**

Figure 10: Comparison of the change in frequency of each CPU core during a 30s execution of Sphinx.



(a) *BaseFreq*=0.4 and *ScalingCoef*=1.0 **(b) *BaseFreq*=0.5 and *ScalingCoef*=0.75** **(c) *BaseFreq*=0.6 and *ScalingCoef*=0.5**

Figure 11: Comparison of the change in frequency of each CPU with fixed parameters settings during a 50ms execution of Xapian.

while *ScalingCoef* is more related to the long-tail distribution of request processing time. For a small portion of requests with longer processing times, a higher *ScalingCoef* is effective in avoiding request timeouts.

5.5 Overhead

Since the neural networks in DeepPower are all lightweight, the DRL agent can be trained without GPU. The parameters updating of the DDPG training algorithm costs 13ms when the batchsize is 64. During testing, DeepPower generates an action in less than a millisecond. Considering one second interval between each step of the DRL, the training and inference overhead is negligible. The number of parameters in the actor neural network is 2096, so the memory and storage overhead is slight. Setting the frequency for a CPU core consumes less than 10us in the thread controller, which enables granular frequency scaling.

The overhead of DeepPower framework is validated. We first run the latency-critical application with a fixed CPU frequency and record the power consumption without DeepPower. Then we run the components of DeepPower while setting the frequency to the previous value each time, and another power consumption is recorded. Then the average difference over multiple intervals is considered as the additional power consumption of DeepPower. The results show that DeepPower delivers 2.81W of additional power

consumption. Compared with the significant power savings from DeepPower, this overhead is acceptable.

6 RELATED WORK

Many studies have concentrated on DVFS-based power management for latency-critical systems in datacenters, whose primary challenge is the disproportion in the service times of requests. Adrenaline [10] points out that requests can be classified as long and short based on features. Rubik [12] goes ahead by modeling the latency distribution. In order to avoid SLA violation, Rubik takes the tail of the distribution as the predicted latency. Considering the long-tailed distribution of request service times, this prediction is overestimated. Gemini [31] notices the relationship between the features of requests and the service time in the web search application and uses a neural network for service time prediction. Gemini selects a low frequency of a request and boosts the frequency when the request is going to time out. Retail [5] argues that linear regression is accurate enough for applications in Tailbench [14]. When a request arrives, Retail enumerates all the frequency levels from small to large and stops when the frequency level is large enough to avoid timing out. Instead of relying on request time modeling, DeepPower utilizes fine-grained frequency scaling techniques to overcome the issue of uneven request processing times, achieving superiority over previous approaches.

Moreover, there exist power management methodologies that utilize the sleep states. Entering the sleep state significantly reduces the power consumption of a core, but returning it to normal state takes a considerable amount of time (i.e. about 100us for C6 state). As a result, utilizing the sleep state carries the risk of request timeouts. DynSleep [7] postpones the requests processing while ensuring tail latency constraints are met exactly. A longer idle period is gained with this delay, and deeper C-state is leveraged to save more power. uDPM[6] is an extension of Rubik, which combines DVFS and sleep states technologies. Utilizing statistical models, uDPM calculates the optimal wake-up time and frequency for each request to maximize power efficiency. However, these methods did not successfully resolve the issue of imbalanced request processing time. The integration of sleep states into our methods represents a significant challenge. We leave this to future work.

7 CONCLUSION

We have presented DeepPower, a DRL-based power management method for latency-critical applications. To overcome the obstacle between the short service time of requests and the long inference time of neural networks, we design a hierarchical control mechanism. DeepPower achieves fine-grain control and has the adaptability to dynamic workloads, which delivers better power savings. Unlike previous methods, DeepPower does not need features of requests, which makes it more general. Experimental results show that DeepPower reduces power consumption by up to 28.4% compared to state-of-the-art methods and decreases the rate of request timeouts.

8 ACKNOWLEDGEMENT

The research is supported by the Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515020054), the National Natural Science Foundation of China (No.62272495), and the Fundamental Research Funds for the Central Universities,Sun Yat-sen University (No. 22qntd1004). The corresponding author is Pengfei Chen.

REFERENCES

[1] [n. d.]. E-commerce search benchmark. https://github.com/alibaba/eCommerceSearchBench.
[2] Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 40, 12 (2007), 33–37.
[3] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 191–205.
[4] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 107–120.
[5] Shuang Chen, Angela Jin, Christina Delimitrou, and José F Martínez. 2022. ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 155–168.
[6] Chih-Hsun Chou, Laxmi N Bhuyan, and Daniel Wong. 2019. μDPM: Dynamic power management for the microsecond era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 120–132.
[7] Chih-Hsun Chou, Daniel Wong, and Laxmi N Bhuyan. 2016. 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*. 212–217.
[8] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.
[9] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. 2018. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905* (2018).
[10] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas F. Wenisch, Jason Mars, Lingjia Tang, and Ronald G. Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. *high-performance computer architecture* (2015).
[11] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2013. Adaptive parallelism for web search. *european conference on computer systems* (2013).
[12] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: fast analytical power management for latency-critical systems. *international symposium on microarchitecture* (2015).
[13] Harshad Kasture and Daniel Sanchez. 2014. Ubik: efficient cache sharing with strict qos for latency-critical workloads. *architectural support for programming languages and operating systems* (2014).
[14] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.
[15] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. *high-performance computer architecture* (2008).
[16] Young Geun Kim and Carole-Jean Wu. 2020. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1082–1096.
[17] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv: Learning* (2015).
[18] Yanpei Liu, Guilherme Cox, Qingyuan Deng, Stark C Draper, and Ricardo Bianchini. 2016. Fastcap: An efficient and fair algorithm for power capping in many-core systems. In *2016 IEEE International symposium on performance analysis of systems and software (ISPASS)*. IEEE, 57–68.
[19] David Lo, Liqun Cheng, Rama K. Govindaraju, Luiz Andre Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. *international symposium on computer architecture* (2014).
[20] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*. 319–330.
[21] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: architectural support for deep sleep. *architectural support for programming languages and operating systems* (2012).

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* (2015).

[23] Ciamac C Moallemi and Mehmet Saglam. 2010. The cost of latency. *SSRN eLibrary* (2010).

[24] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*. 1–13.

[25] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash N. Ananthakrishnan, and Eliezer Weissmann. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* (2012).

[26] Eric Schurman and Jake Brutlag. 2009. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. oreilly.

[27] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. United states data center energy usage report. (2016).

[28] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[29] Ratnala Vinay, Pradip Sasmal, Chandrajit Pal, Toshihisa Haraki, Kazuhiro Tamura, Chirag Juyal, Mohamed Amir Gabir Elbakri, Sumohana Channappayya, and Amit Acharyya. 2022. Light Weight RL Based Run Time Power Management Methodology for Edge Devices. In *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 1–4.

[30] Yiming Wang, Weizhe Zhang, Meng Hao, and Zheng Wang. 2021. Online power management for multi-cores: A reinforcement learning based approach. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 751–764.

[31] Liang Zhou, Laxmi N. Bhuyan, and Kadangode K. Ramakrishnan. 2020. Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines. *international symposium on microarchitecture* (2020).

[32] An Zou, Karthik Garimella, Benjamin Lee, Christopher Gill, and Xuan Zhang. 2020. F-LEMMA: Fast learning-based energy management for multi-/many-core processors. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 43–48.