Yawn: A CPU Idle-state Governor for Datacenter Applications

Erfan Sharafzadeh[1,2], Seyed Alireza Sanaee Kohroudi[1]
Esmail Asyabi[1], Mohsen Sharifi[1]

[1]Iran University of Science and Technology [2]Johns Hopkins University
e.sharafzadeh@jhu.edu,{sarsanaee,e_asyabi}@comp.iust.ac.ir,msharifi@iust.ac.ir

ABSTRACT

Idle-state governors partially turn off idle CPUs, allowing them to go to states known as idle-states to save power. Exiting from these idle-sates, however, imposes delays on the execution of tasks and aggravates tail latency. Menu, the default idle-state governor of Linux, predicts periods of idleness based on the historical data and the disk I/O information to choose proper idle-sates. Our experiments show that Menu can save power, but at the cost of sacrificing tail latency, making Menu an inappropriate governor for data centers that host latency-sensitive applications. In this paper, we present the initial design of Yawn, an idle-state governor that aims to mitigate tail latency without sacrificing power. Yawn leverages online machine learning techniques to predict the idle periods based on information gathered from all parameters affecting idleness, including network I/O, resulting in more accurate predictions, which in turn leads to reduced response times. Preliminary benchmarking results demonstrate that Yawn reduces the 99^{th} latency percentile of Memcached requests by up to 40%.

KEYWORDS

Operating System, Power Management, Tail Latency, Idle-state Governor

ACM Reference Format:
Erfan Sharafzadeh[1,2], Seyed Alireza Sanaee Kohroudi[1], Esmail Asyabi[1], Mohsen Sharifi[1]. 2019. Yawn: A CPU Idle-state Governor for Datacenter Applications. In *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3343737.3343740

APSys '19, August 19–20, 2019, Hangzhou, China

ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343740

1 INTRODUCTION

In-memory key-value databases are the beating heart of the current data centers. The workloads for such databases consist of short-lived tasks generated initially from a user-facing server. User requests are partitioned into multiple queries distributed among servers in data centers [11]. The aggregation of query results forms the final response to the user request and the latency of the slowest server determines the overall QoS [3, 10].

Energy efficiency, on the other hand, is a highly desirable goal for data centers as it determines the total cost of ownership [5]. Studies have shown that considering the energy delivery losses, reducing a single watt energy consumption in processors can thus turn up to 3x Watts of data center energy savings [15]. Since the utilization of servers in data centers follows a periodic pattern [2], it is likely that several CPU cores become idle many times a day. This provides an opportunity to enter CPU idle-states. In spite of the fact that governing the idle-states curb the power consumption of CPUs, exiting from these states can be notably time-consuming [21, 30].

We find out that Menu, the default idle-state governor of Linux, does not consider the incoming network request rate (network I/O) as a parameter in the prediction of task arrival times. This results in inaccurate predictions regarding network-intensive workloads, imposing notable exit latency of idle-states, and worsening the request tail latency [18, 36].

Additionally, the default task load-balancer of Linux prefers to put the newly awakened tasks on idle cores to accelerate their execution. We believe this policy not only increases the power consumption but also often lengthens the tail latency. This is because of the fact that idle cores residing in deep idle-states need to spend a considerable amount of time relative to the response time of a network request to become operational again, while it might be more efficient to let the

active processor cores enqueue the task for future execution and allow the sleeping cores save more power [4].

In this paper, we present an idle-state governor called Yawn, specifically designed for data centers that host latency-sensitive workloads. The main goal of Yawn is to mitigate the tail latency of network-intensive workloads without compromising power consumption. We leverage "prediction with expert advice", an online machine learning approach to predict the arrival times of tasks for the idle-state governor. Also, Yawn's power-aware load-balancer avoids running newly awakened tasks on idle cores if there is an operating core capable of accommodating it.

The results of our initial experiments with Yawn prototype demonstrate that Yawn can improve the tail latency of Memcached requests by near 50% in the improvement window -that is the difference between the response time in the default c-state settings and when we disable c-states. Also considering the tradeoff between latency performance and power efficiency, Yawn tries to keep its power consumption lower than Menu at all times.

2 BACKGROUND AND MOTIVATION

2.1 CPU Power-saving and Idle-states

When a CPU is in an idle-state and a task becomes ready for execution, it takes a while for the CPU to activate its components again. This duration is referred to as *exit latency*. A deeper idle-state imposes higher exit latency. This exit latency can be a source of long-tailed response times in I/O-sensitive tasks. We seek a management strategy that chooses the most suitable idle-state for an idle CPU that does not hurt the performance of running tasks. Such strategies are managed by a component called idle-state governor. A clairvoyant idle-state governor knows the arrival times of the next ready tasks. Consequently, it chooses the deepest possible idle-state, resides there adequately and finally returns to the operational state right before the arrival of the next ready task.

Idle-state governors are one of the few components of the operating systems that can borrow machine learning techniques in order to provide accurate predictions. At the very least, having a clairvoyant governor is impracticable since the arrival of tasks depends on many internal and external factors, leaving no choice but resorting to governors that leverage heuristic approaches to predict the sleep intervals. This motivates us to integrate online machine learning schemes into the operating system.

Figure 1 presents an overview of the cpuidle subsystem operation. When a CPU core is in a sleep state, an interrupt or a write to the memory region -storing the flag that indicates the availability of ready tasks for the scheduler- can initiate the exit from the sleep state (1). The cpuidle driver

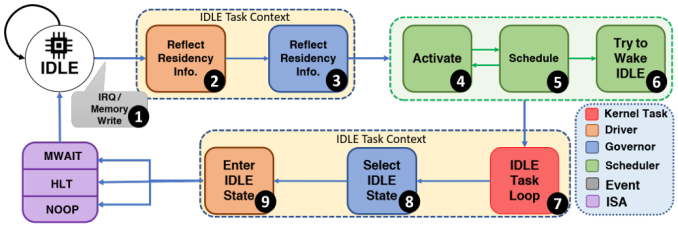


Figure 1: The idle-state management subsystem operation overview

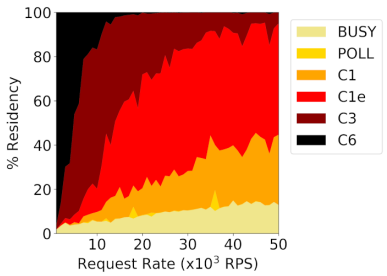


Figure 2: The c-state residency of the CPU cores under different Memcached request rates

then updates the last entered state and the residency time (2) followed by a similar call, the active governor updates its statistics (3). Then the ready task, which might be a kernel worker or a user application, becomes activated (4) and eventually scheduled for the execution (5). This cycle is repeated until the scheduler, forcing it to wake up the idle task (6), returns no ready task. The idle task contains an endless loop calling the corresponding governor to select an appropriate idle-state (7). The governor embodies the actual policies of idle-state selection, chooses a proper state, and returns it to the idle task (8). The driver then executes the selected idle-state (9) by translating it to machine instructions such as *MWAIT*, for deeper states that support memory monitoring during sleep, *HLT*, an older facility for shallow sleeps and a nop loop if the system decides to remain active [13, 19, 34].

2.2 Linux Idle-state Governor

Menu, the current idle-state governor of Linux, consists of two main components: selection and reflection. Menu's selection algorithm selects a correction factor (CF) from a histogram of twelve integer factors, based on the remaining time until the next timer event. These factors are updated in the reflection step and demonstrate the closeness of the next predicted wake-up to the next timer tick. The multiplication of the factor and the upcoming timer tick gives the foremost prediction of the following wake-up event. To enhance the prediction further, Menu also uses a simple moving average of the latest residency times. In the case of a low standard

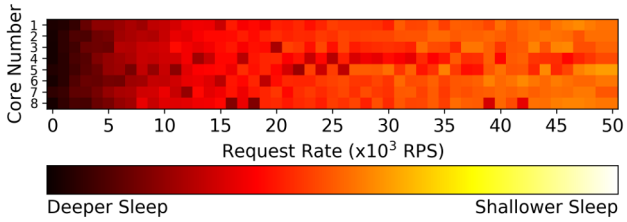
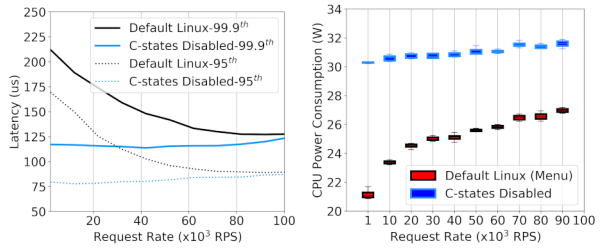


Figure 3: C-state residencies in different Memcached request rates in a Linux system



(a) Tail latency **(b) Power consumption**

Figure 4: The latency and power consumption gaps between the default Linux governor and disabled idle-states

deviation, the moving average is used instead as the prediction. Finally, based on the number of processes waiting for a disk I/O and the run-queue load, the prediction is updated.

The reflection process consists of updating the governor's data structures and occurs before selecting the next idle-state. The governor retrieves the last entered c-state and its residency time and updates the corresponding correction factor using an exponential moving average.

Unfortunately, Menu does not directly account for network I/O in its prediction of sleep intervals. In fact, it only considers the tasks' waiting time for disk I/O as a factor to reduce the predicted value. This inevitably causes premature wake-ups and thus latency hiccups, leading to unpredictable network request latency. [23, 27–29, 32].

Figure 2 shows the c-state residencies of CPU cores as the load on the Memcached server increases. As the governor chooses the deepest sleep state in lower request rates, requests in these rates experience notable delays due to high idle-state exit latency. Then, while the rate increases, Menu governor chooses shallower idle-states, which are less harmful to the request latency. The load-balancing behavior of the system is also presented in Figure 3. It shows that the CPU cores are rather equally utilized and the attempts of the task load-balancer in trying to save more power on idlest cores by keeping them vacant is not reflected. That is because the scheduler always chooses an idle core for a newly woken task rather than putting the task on its ready task queue (run-queue) for each active CPU core.

From a motivational perspective based on figures 4a and 4b, we can observe that the maximum latency improvement is determined by the highest c-state exit latency, which means that if we disable the governor, the tail latency is reduced by up to the highest exit latency.

3 DESIGN

Yawn is a generic platform that is consisted of an idle-state governor and a process load balancer integration, working together to provide a better balance between the latency performance and power efficiency of the operating system. The idle-state governor comprises independent sleep-duration predictors and acts to aggregate their predictions and weight them based on their accuracy. The load balancer uses the data from Yawn's governor and the process scheduler to decide on task placements. Yawn's design overview is depicted in Figure 5.

3.1 The Idle-state Governor

Each of Yawn's predictors (also referred to as experts) is designed to predict the next sleep duration based on its gathered information (e.g., network request arrivals). Yawn's governor calculates the weighted average of the forecasters' predictions to pick a proper c-state and then estimates the time the CPU should reside in that state. After the actual arrival time is revealed, the governor updates experts' weights. This approach is called *Prediction with Expert advice* in online machine learning, and brings about the advantage of supporting various configurations.

There are various reasons for a process to wake up, like the availability of an I/O event or the process scheduler's decision. Waking up a process mandates a set of actions from the operating system to be done. These include background operations such as context switching or allocating proper memory regions. It is also worth mentioning that since these operations usually occur at a high frequency, their performance impact will be drastic at scale [39]. Idle-state governor decides while keeping an eye on all these tasks. We believe that each component that could be impacted by the idle-state governor and forgoes the power efficiency should be scrutinized and is a potential *Expert* for the CPU idle-state governor in our design.

Experts are designed, suited for batched, critical, disk I/O intensive or offline workloads, and the system will autonomously adapt the expert weights, therefore favoring the most appropriate experts. Before getting to the details of Yawn experts, we describe *prediction with expert advice* in more detail.

3.2 Prediction with Expert Advice

Prediction with expert advice is used when there are multiple sources that predict the event of interest independently. At each time unit t, each source that is theoretically called an

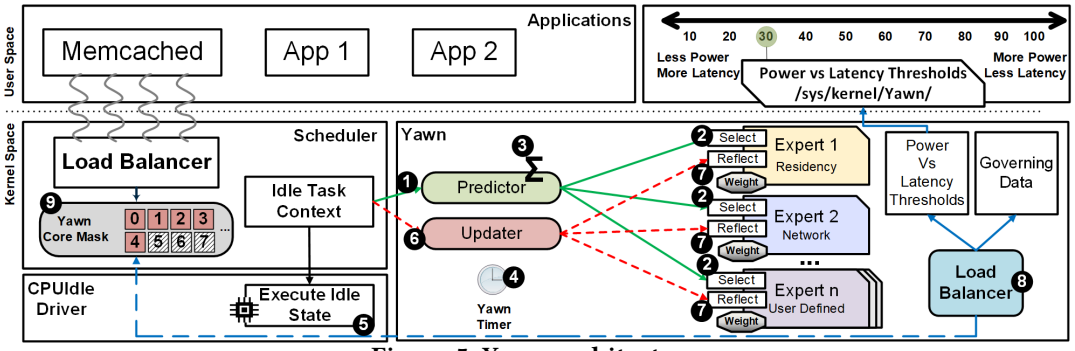


Figure 5: Yawn architecture

expert E (also oracle) predicts the next outcome denoted by $f_{E,t}$. The forecaster then uses a weighted average of the experts' predictions as the final decision at time t as denoted by Equation 1.

$$\hat{p}_t = \frac{\sum_{i=1}^{N} w_{i,t-1} f_{i,t}}{\sum_{i=1}^{N} w_{i,t-1}} \tag{1}$$

wherein N is the number of available experts and $w_{i,t}$ indicates the weight of expert i at time t.
Finally, when the environment reveals the real outcome, the forecaster updates the weights of the experts according to their success in their estimations. The distance function of each expert's forecast from the real event is called the loss function denoted by $\ell_{i,t}$ and the ultimate goal of the forecaster is to minimize the *cumulative Regret* of each expert that is defined as the difference between the forecaster's cumulative loss and the expert's cumulative loss after n rounds of prediction. According to an exponential weight-update function described in [6], the weight of expert i at time t can be derived as in Equation 2.

$$w_{i,t} = \frac{w_{i,t-1} e^{-\eta \ell(f_{i,t}, y_t)}}{\sum_{j=1}^{N} w_{j,t-1} e^{-\eta \ell(f_{j,t-1}, y_t)}} \tag{2}$$

3.3 Idle-state Selection and Reflection

Figure 5 depicts the overall design principles of Yawn. When the OS scheduler finds no available ready task to execute, the context is switched to the idle task that consists of an endless loop that calls the cpuidle subsystem to select an appropriate idle-state (1). In the provided selection interface, Yawn queries all registered experts to predict next sleep duration of the current processor (2), then uses a weighted average as the conciliation of all experts (3). The selected c-state is the deepest state that satisfies the target residency constraints based on the derived weighted average. Before executing the idle-state, if the corresponding expert detects network activity, a timer is armed to wake the CPU core at the predicted time spot (4). The timer decreases the chance that an incoming request will have to wait for a full exit latency.

The chosen state is then executed (5) and eventually an event awakes the system revealing the actual duration of sleep to the governor (6). To extend the accuracy of the system, the governor updates the corresponding weight of the experts using an exponential weight update function and finally provides the experts with a chance to update their data with the actual sleep residency time (7). It is worth mentioning that the wake-ups by the pre-armed Yawn timer will not count as an event of interest for the governor, so in case of a premature wake-up, skipping the update, the system will be put back to sleep again.

3.4 The Residency Expert

In online machine learning, the moving average algorithm proves to provide a decent resolution with a sustainable complexity [6]. Therefore, for the initial phase of Yawn's implementation, we utilize the exponential moving average (EMA) of the last 8 idle-state residency durations according to Equation 3:

$$EMA_{t+1} = 18 \times EMA_t + 2 \times y_t \tag{3}$$

wherein y_t is the actual sleep duration of the current CPU core. Yawn's residency expert always predicts the next sleep duration using the existing history gathered during the past prediction rounds.

3.5 The Network Expert

To oversee the effect of network requests on the governor, we require an independent expert that infers per-CPU incoming request rate and uses this rate to predict the next idle-state residency. With the arrival of each network request, an interrupt is raised by the NIC to invoke the network driver's receive routine. Then three scheduling events are triggered consecutively to (1) process the incoming packet, (2) execute the user application and (3) send the response packet.

Using the deduced event rate, the network expert predicts the next sleep duration, and if it doesn't detect any network events, simply refuses to vote.

3.6 The Load Balancer

Yawn's timer mechanism can possibly increase the number of wake-ups. Increased number of wake-ups mitigates response times of network requests, but it prevents idle-cores from staying in deep idle-states long enough, resulting in higher power consumption. Therefore, we propose a new load balancer that tries to avoid running incoming tasks on idle cores if there is an operating core that can accommodate them. In this way, when a task is woken up, it finds an operating CPU-core to execute the new task. If it cannot find an operating core that can accommodate the new task, it turns on a neighboring idle-core instead.

The advantages of such a technique are twofold. First, it gives enough time to idle cores to remain/go in/to deep idle-states to save more power. Secondly, executing newly awakened tasks on operating cores eliminates the impact of high exit latency caused by idle-cores, thereby shortening response times and tail latency of I/O-intensive workloads.

4 EVALUATION

4.1 Setup

We evaluate an initial implementation of Yawn in an environment consisting of four production-level machines. The server machine has two Intel Xeon E5-2697 v2 processors and 64GB of memory, alongside two 10GbE network interfaces with Receive-side scaling enabled. Processors possess 12 physical cores that can operate up to 2.7 GHz and support four c-states with exit latency of 1 to 82μs. The operating system is Ubuntu 16.04 with modified Linux kernel 4.4.120. The other three machines cooperate to generate load on the server. We activated the *performance* P-state governor on all the machines and disabled Intel Turbo Boost and Hyper-threading to prevent unwanted performance degradations.

We deployed Memcached [14] on the server and configured its worker threads to match the number of available cores on the system. To generate the workload, we used *Mutilate* [26], a distributed Memcached load generator, used in related works to simulate realistic Facebook-like workloads. We configured Mutilate using its predefined Facebook distributions.

4.2 Preliminary Results

First, to show how the load balancer operates, we repeat the experiment presented in Section 2. The idle-state residency heat map of CPU cores, presented in Figure 6, shows how Yawn distributes tasks among CPU cores as the load (incoming network rate) increases. According to the heat-map, at first, only one CPU core is in the operational mode, ready to serve requests while all other CPU cores are in a deep idle-state, saving power. As the rate increases, Yawn's load balancer will elastically turn on more CPU cores if needed and lets other CPU cores to remain in deep idle-states.

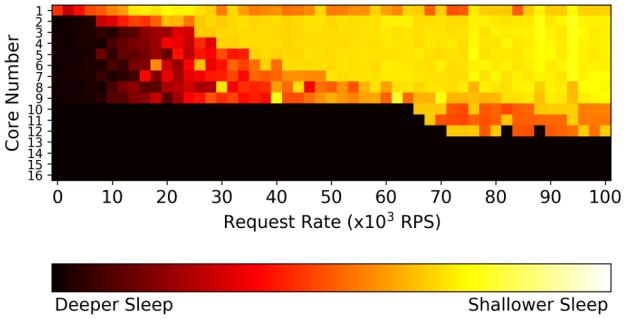
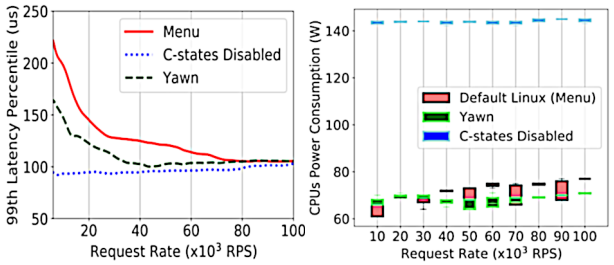


Figure 6: C-state residencies in different Memcached request rates for the proposed load balancing mechanism



(a) Tail latency **(b) Power consumption**

Figure 7: Evaluation of Yawn for NUMA systems

Figure 7a depicts the 99^{th} percentiles of the Memcached response-time distribution under Yawn, the default governor of Linux (Menu) and when the idle-states are completely disabled. We can observe that using a combination of *Prediction with expert advice*, as a more accurate prediction model and the proposed load-balancing scheme, up to tail latency improvements can be achieved. This is because, Yawn's governor estimates the arrival times of network requests more accurately by turning the corresponding cores on just before requests arrive, therefore avoiding long exit latency of deep idle-states. Moreover, Yawn's load balancer further mitigates tail latency because it prefers to execute newly awakened tasks on operating cores rather than on idle cores, avoiding exit latency of idle cores.

Figure 7b shows the effectiveness of Yawn's load balancer on decreasing the power consumption of the processors. It demonstrates that the aggregated energy consumption of the two processors can be reduced significantly because Yawn keeps unutilized CPU cores in deep idle-states and it does not use the idle cores as long as the existing operating CPU cores can accommodate the incoming network load. Similar results can also be seen from Figure 8, which presents the results from running a varying Memcached trace over a longer period of time.

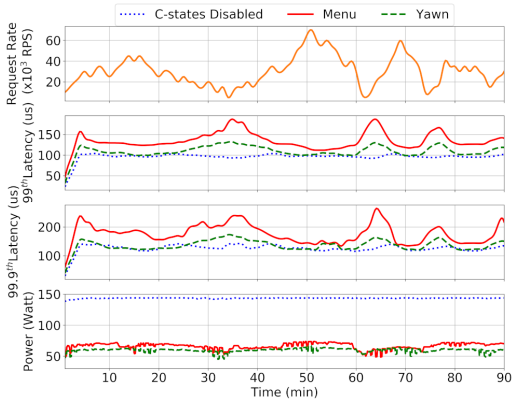


Figure 8: Evaluating Yawn under dynamic loads

4.3 Discussion

The above results, although on a narrow application domain, present a promising start for rethinking the design of idle-state management. Similar attempts have also been made by the Linux community in the recent version of the Linux kernel [1]. Choosing Memcached as the primary benchmark was merely due to its widespread attention in the related literature and also in the industry. Nevertheless, to make Yawn a general-purpose idle-state governor, various types of workloads need to be evaluated.

Moreover, the initial design of two experts will be complemented by supporting all types of threading models and also by considering experts that can be used in relatively less critical environments (e.g., HPC systems, workstations, and data-driven systems). We will also consider other learning techniques to find the answer to whether *expert advice* is the ultimate mean to settle the tradeoff between latency and power consumption in idle-state management.

5 RELATED WORK

Processor Power Management. The energy consumption of a server machine can be mainly attributed to processors, memory and I/O peripheral devices, among which, the processor's share is about a third of the total power [31]. Operating systems adjust the power utilization of CPUs through hardware-defined interfaces called c-states, for idle-state management and p-states, for frequency scaling while a part of arbitration is performed directly by the hardware. Governing the frequency of the processors plays an essential role in controlling the energy consumption of servers under load since the idle-states are not invoked when the server load increases. Several works have investigated frequency governors [17, 20, 23–25, 35, 37, 38, 40].

Governing Idle-states. Idle-state Governors aim to predict the sleep duration of the system. Meisner [31] introduces transitions between full system sleep and operational state to save more energy while considering the latency requirements. This strategy is further enhanced by utilizing request batching [33]. Q-Learning is used in [36] to give a better prediction of CPU sleep durations, this study further claims that Menu governor adopts slowly to the workload pattern changes.

In a recent work, Ilsche [18] has found inaccurate predictions in Menu that lead to the selection of shallower states. He calls such mistakes *Powernightmares* and uses high-resolution timers as a fall-back mechanism to deal with them. The study has only examined these inaccuracies for high-performance computing workloads. Duan [12] has proposed a model-based prediction of idle durations that is relied on performance feedback.

Application-level Power Management. The impact of idle-states on Memcached has recently been investigated in [43]. The authors have proposed a core arbitration algorithm that decides to turn on/off the processor cores based on the incoming request rate and the latency of the previous Memcached requests. [9] has targeted the energy consumption by postponing the Memcached requests until a user-defined tail latency cap, allowing the idle cores to further benefit from deep sleeps. Query processing time prediction and request batching are also studied in server clusters [7, 8, 16, 22, 41, 42]. Such strategies allow the idle servers to enter package c-states and significantly save more power.

6 CONCLUSION

We presented Yawn, an idle-state governor to mitigate tail latency without increasing the power consumption. Yawn leverages an online machine learning technique where each parameter affecting periods of idleness is defined as an expert that can predict idleness periods. Our initial experiments show that Yawn can reduce the tail latency of a Memcached workload by up to 40%. Yawn is scalable regarding the number of experts predicting idleness, allowing system administrators to define more experts for more accurate predictions based on desired QoS.

Further, in Yawn, we have designed a load balancer to bridge the gap between idle-state governors and task load-balancers. Our energy-aware load-balancer avoids running newly awakened tasks on idle cores if possible, allowing idle cores to go to deeper idle-states to save more power. Our future direction will include the presentation of Yawn's formal definitions, alongside an extensive evaluation of Yawn using common data center workloads.

REFERENCES

[1] 2019. Linux Kernel Mailing List. https://lkml.org/lkml/2019/1/6/178. Accessed: 2019-04-17.

[2] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 405–417.

[3] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

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

[5] Luiz André Barroso and Urs Hölzle. 2009. The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines. *Synthesis Lectures on Computer Architecture* 4, 1 (2009), 1–108.

[6] Nicolo Cesa-Bianchi and Gabor Lugosi. 2006. *Prediction, Learning, and Games*. Cambridge University Press.

[7] Chih-Hsun Chou, Laxmi N Bhuyan, and Shaolei Ren. 2017. TailCut: Power Reduction under Quality and Latency Constraints in Distributed Search Systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 1465–1475.

[8] 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)*. 120–132.

[9] 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 (ISLPED '16)*. ACM, New York, NY, USA, 212–217.

[10] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.

[11] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. (2019), 79–94. https://www.usenix.org/conference/nsdi19/presentation/didona

[12] L Duan, D Zhan, and J Hohnerlein. 2015. Optimizing Cloud Data Center Energy Efficiency via Dynamic Prediction of CPU Idle Intervals. In *2015 IEEE 8th International Conference on Cloud Computing*. ieeexplore.ieee.org, 985–988.

[13] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking Energy. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 393–406.

[14] Brad Fitzpatrick and Anatoly Vorobey. 2011. Memcached: a Distributed Memory Object Caching System.

[15] Corey Gough, Ian Steiner, and Winston Saunders. 2015. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress.

[16] Vishal Gupta, Paul Brett, David A Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan, and Ganapati Srinivasa. 2012. The Forgotten 'Uncore': On the Energy-Efficiency of Heterogeneous Cores. In *2012 USENIX Annual Technical Conference*. usenix.org, 367–372.

[17] C H Hsu, Y Zhang, M A Laurenzano, D Meisner, T Wenisch, J Mars, L Tang, and R G Dreslinski. 2015. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 271–282.

[18] Thomas Ilsche, Marcus Hähnel, Robert Schöne, Mario Bielert, and Daniel Hackenberg. 2018. Powernightmares: The Challenge of Efficiently Using Sleep States on Multi-core Systems. In *Euro-Par 2017: Parallel Processing Workshops*. Springer International Publishing, 623–635.

[19] Intel. 2018. *Intel Software Developer's Manual*.

[20] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, 519–532.

[21] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. 2014. Tradeoffs Between Power Management and Tail Latency in Warehouse-scale Applications. (2014), 31–40. https://doi.org/10.1109/IISWC.2014.6983037

[22] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 615–629.

[23] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 598–610.

[24] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. 2018. Workload-Aware CPU Performance Scaling for Transactional Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 291–306.

[25] Etienne Le Sueur and Gernot Heiser. 2011. Slow Down or Sleep, That Is the Question. In *2011 USENIX Annual Technical Conference*.

[26] Jacob Leverich. 2014. Mutilate: High-performance Memcached Load Generator.

[27] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, 4:1–4:14.

[28] Jialin Li, Naveen Kr Sharma, Dan R K Ports, and Steven D Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, 9:1–9:14.

[29] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 301–312.

[30] Yanchao Lu, Quan Chen, Yao Shen, and Minyi Guo. 2017. Electro: Toward QoS-Aware Power Management for Latency-Critical Applications. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 221–228.

[31] David Meisner, Brian T Gold, and Thomas F Wenisch. 2009. PowerNap: Eliminating Server Idle Power. *SIGARCH Computer Architecture News* 37, 1 (March 2009), 205–216.

[32] 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 (ISCA '11)*. ACM, New York, NY, USA, 319–330.

[33] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: Architectural Support for Deep Sleep. *SIGARCH Comput. Archit. News* 40, 1 (March 2012), 313–324. https://doi.org/10.1145/2189750.2151009

[34] V Pallipadi, S Li, and A Belay. 2007. cpuidle: Do Nothing, Efficiently. *Proceedings of the Linux Symposium* (2007).

[35] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 342–355.

[36] Andrei Roba and Zoltan Baruch. 2015. An Enhanced Approach to Dynamic Power Management for the Linux Cpuidle Subsystem. In *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), 2015* . 511–517.

[37] R Sen and A Halverson. 2017. Frequency Governors for Cloud Database OLTP Workloads. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6.

[38] Rathijit Sen and David A Wood. 2017. Pareto Governors for Energy-Optimal Computing. *ACM Trans. Archit. Code Optim.* 14, 1 (March 2017), 6:1–6:25.

[39] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 177–194.

[40] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *2016 USENIX Annual Technical Conference*. usenix.org, 309–322.

[41] Fan Yao, Jingxin Wu, Suresh Subramaniam, and Guru Venkataramani. 2017. WASP: Workload Adaptive Energy-latency Optimization in Server Farms Using Server Low-power States. In *IEEE 10th International Conference on Cloud Computing (CLOUD), 2017*. 171–178.

[42] Fan Yao, Jingxin Wu, Guru Venkataramani, and Suresh Subramaniam. 2017. TS-Bat: Leveraging Temporal-Spatial Batching for Data Center Energy Optimization. In *IEEE Global Communications Conference (GLOBECOM 2017)*. 1–6.

[43] Xin Zhan, Reza Azimi, Svilen Kanev, David Brooks, and Sherief Reda. 2017. Carb: A C-state Power Management Arbiter for Latency-critical Workloads. *IEEE Computer Architecture Letters* 16, 1 (2017), 6–9.