# Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines

Liang Zhou, Laxmi N. Bhuyan, K. K. Ramakrishnan
*Computer Science and Engineering Department*
*University of California Riverside, USA*
lzhou008@ucr.edu, {bhuyan, kk}@cs.ucr.edu

***Abstract*—Saving energy for latency-critical applications like web search can be challenging because of their strict tail latency constraints. State-of-the-art power management frameworks use Dynamic Voltage and Frequency Scaling (DVFS) and Sleep states techniques to slow down the request processing and finish the search just-in-time. However, accurately predicting the compute demand of a request can be difficult. In this paper, we present Gemini, a novel power management framework for latency-critical search engines. Gemini has two unique features to capture the per query service time variation. First, at light loads without request queuing, a two-step DVFS is used to manage the CPU power. Our two-step DVFS selects the initial CPU frequency based on the query specific service time prediction and then judiciously boosts the initial frequency at the right time to catch-up to the deadline. The determination of boosting time further relies on estimating the error in the prediction of individual query's service time. At high loads, where there is request queuing, only the current request being executed and the critical request in the queue adopt a two-step DVFS. All the other requests in-between use the same frequency to reduce the frequency transition overhead. Second, we develop two separate neural network models, one for predicting the service time and the other for the error in the prediction. The combination of these two predictors significantly improves the power saving and tail latency results of our two-step DVFS. Gemini is implemented on the Solr search engine. Evaluations on three representative query traces show that Gemini saves 41% of the CPU power, and is better than other state-of-the-art techniques.**

***Index Terms*—power management, DVFS, search engine, neural network, service time prediction**

## I. Introduction

Power management in data centers is challenging because of the fluctuating workloads and strict task completion time requirements. Web search is an example of latency critical applications, whose tail latency is critical to a data center operator's revenue [1]. Aggregation policies [2]–[4] in web search enforce strict Service Level Agreements (SLAs) on the tail latency of search requests at each Index Serving Node (ISN) server [5], [6], to avoid search quality and response time degradations [7], [8]. In order to meet tight latency constraints, ISN server utilizations for search engines are typically kept low [9]–[11]. But, lightly loaded ISN servers waste a lot of energy in the data center [12], [13]. This has prompted plenty of prior research [14]–[18] to save power for latency-critical search engines.

Most of this research has been based on two broad techniques: Dynamic Voltage and Frequency Scaling (DVFS) and Sleep states. DVFS schemes such as Pegasus [14] and Rubik [18] dynamically adjust the CPU's frequency to save power. Pegasus is a course-grained epoch-based policy suitable to reduce power for long-term variability in the workload. Rubik, on the other hand, is a query based policy that captures short-term variability while guaranteeing that the constraint on the high percentile latency is not exceeded. Sleep states frameworks such as PowerNap [9] and DreamWeaver [11] put the server into sleep during idle periods. On the other hand, $\mu$DPM [17] adjusts both sleep period and DVFS to meet the target latency. In this paper, we focus on using DVFS, although the technique can also be extended to Sleep states. The main challenge lies in the fact that search requests' latencies have both short and long term variations and a request's computation requirement (i.e., total CPU cycles) cannot be predicted accurately [16], [18]. Hence, maintaining a balance between energy savings and meeting deadlines is the goal of our work.

In this paper, we present Gemini, a fine grained DVFS scheme with accurate per-query service time prediction. Our latency prediction is based on a sophisticated Neural Network (NN) model that captures realistic search engine characteristics. Even then, it is unrealistic to have 100% accuracy. Hence, Gemini proposes a heuristic approach, using a two step DVFS, to achieve deadline constraints through frequency boosting. In the first step upon a request arrival, Gemini selects an initial CPU frequency according to the predicted service time based on the NN model. In the second step, depending on the query progress at a particular time, Gemini judiciously boosts the CPU frequency, to catch up with the request's deadline, if the request processing is lagging. The boosted frequency is set to the maximum core frequency, but the boosting time must be determined accurately. In our model, this time is calculated based on a second NN error predictor for the latency estimation. Based on these two separate NN predictors, Gemini automatically adjusts a request's initial frequency and boosting time to accommodate the query specific variations.

Our paper has some similarity to the step-wise DVFS scheme proposed in PACE [19]. However, the theoretical framework they proposed is difficult to implement in practice. PACE still relies on a distribution to sample the residual work for a request. Additionally, its per-query Linear Programming (LP) model has a very high overhead, precluding real deployment. Another scheme, proposed in EETL [16], is based

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00059

TABLE I
POWER MANAGEMENT COMPARISON

| Schemes | Uncertainty | Unknown Demand | DVFS Control & Implementation | Reconfigure for Critical Request |
|---|---|---|---|---|
| Pegasus [14] | long-term (per epoch) | deadline violation and latency history | centralized feedback based controller | no concept of critical request |
| Rubik [18] | short & long (per request) | tail of latency distribution obtained from server profiling | statistical model in software runtime | ✓ |
| PACE [19] | short & long (per request) | work distribution sampled from recent tasks | simulated (unlimited) step-wise DVFS through per query LP solver | latter request might violate its deadline |
| EETL [16] | long-term (per epoch) | execute request until its time threshold expires | PID controller | latter request might violate its deadline |
| Gemini | short & long (per request) | neural network based latency & error predictors | heuristic one or two-step DVFS in user-space (with only 0.89% extra predict and schedule overhead) | ✓ |

on readjusting the execution speed on Asymmetric Multicore Processors at specific time epochs as predicted through a feedback control circuit. Requests in an epoch share the same boosting threshold without capturing short-term service time variations. However, ours is a query specific approach that captures both the service time and deadline violation for each query. The characteristics of different DVFS approaches including our Gemini are presented in Table I.

Initial estimation of the frequency is very important to satisfy the deadline constraint of a request. A working thread on an ISN server has to score the documents of a query's posting list [20] one by one to retrieve the top-$K$ relevant results [21], [22]. This provides an opportunity to precisely estimate a request's service time. Although some documents on a query's posting list might be skipped due to pruning techniques [23], [24], an advanced prediction model with multiple features can be developed to improve the accuracy [25]. We propose a NN model that incurs only 0.79% additional delay on our platform, which is less than the centralized control overhead in Pegasus [14]. Query specific latency prediction has been proven useful to significantly reduce the 99th percentile tail latency on Microsoft Bing [26]. Even then, this prediction is likely to have some errors. Therefore, in Gemini, we use a second NN model to predict the error. The sum of predicted latency and error estimate for a query is used to determine the correct frequency boosting time in the two-step DVFS scheme.

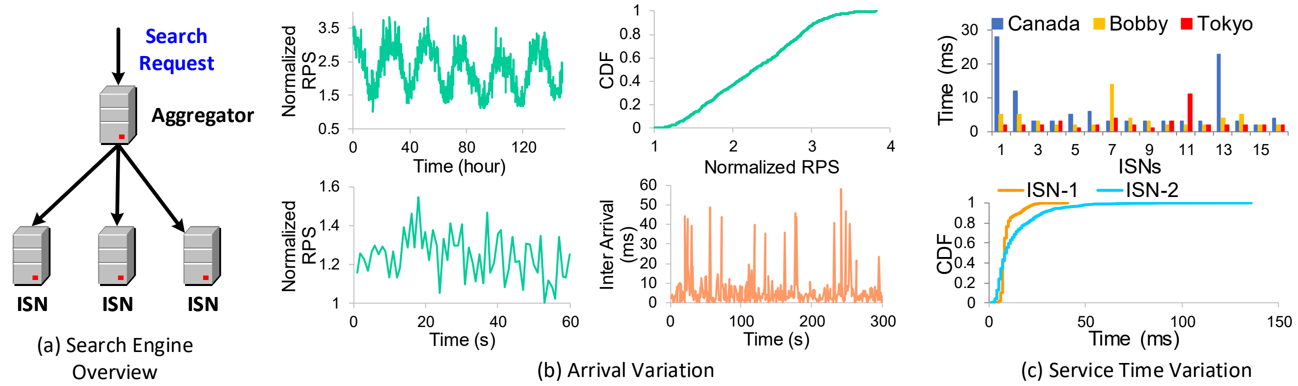
For medium and high server loads with request queuing, we boost the current CPU frequency whenever a critical request arrives. A critical request is the one for which the target latency is violated if we adhere to the current DVFS setting. In the general case, only the current request being executed and the critical request adopt a two-step DVFS. All the other requests in-between execute at the current frequency, to minimize the frequency transition overheads. Requests using the same frequency have already met their latency constraints before the critical request arrives. The arrival of critical request results in a higher current frequency. Thus, requests in-between will finish their tasks before the deadline.

Gemini is implemented in the Solr search engine [27], which is deployed on a 24-core platform with user-space DVFS control and power measurement. Experimental results on three different query traces [8], [16], [28] prove that Gemini saves 41% of the CPU power, which is 1.53 times better than Rubik [18] and 2.05 times better than Pegasus [14]. At the same time, Gemini produces the minimum deadline violation rate compared to Pegasus and Rubik. Our major contributions are the following:

- A novel heuristic two-step DVFS is proposed, which properly boosts the CPU frequency to catch-up to deadlines, on a per-query basis. The frequency steps are adjusted at the arrival and departure of a critical request when there are multiple requests in the queue.
- A low overhead NN model is developed to precisely predict the service time at a per-query granularity based on realistic features in query processing. Additionally, a separate error prediction model is utilized to determine the exact time for boosting the frequency in our two-step DVFS.
- Finally, Gemini is implemented on a real platform using the commonly used Solr search engine. Three representative query traces are evaluated for energy saving and compared with previous techniques.

## II. BACKGROUND AND MOTIVATION

As shown in Fig. 1 (a), a search engine typically employs a partition-aggregate architecture [29]. Data collections in a search engine often contain billions of documents [20], [30], which are partitioned into a number of shards. Typically, one shard of documents is served by an ISN server. The ISN organizes its local documents as an inverted index [31], in which each key in the query term dictionary [32] is linked with a list of matched documents (i.e., the query term's posting list). Whenever the aggregator receives a search request from a client, it will broadcast the request to all the ISNs. Every ISN then searches its local index performing a scoring of documents in the posting list. As there are millions of documents in an index shard, only the top-$K$ scored documents are sent back to the aggregator [33]. Finally, the aggregator merges and ranks all the responses from the ISNs before sending the search results to the client [34]. The overall latency for a search result is limited by the slowest response arriving from the ISNs. Thus, it is critical to meet the strict tail latency constraint at the ISN servers, for good search latency and quality. The stragglers will by ignored by the aggregator of the responses.



(a) Search Engine Overview

(b) Arrival Variation

(c) Service Time Variation

Fig. 1. Search workload exhibits high variations. In part (b), top-left figure presents the diurnal and day-of-week pattern for the normalized Request per Second (RPS); top-right figure gives the corresponding CDF of RPS values; bottom-left figure shows the arrival variations in a short term; bottom-right figure plots the inter-arrival time variations.

### *A. Search Workload Characteristics*

Fig. 1 (b) shows that the incoming request rate to a search engine can exhibit high variations over both long and short time intervals. The top-left sub-figure plots the Request per Second (RPS) rate for a Wikipedia query trace [28], [35] over a period of 150 hours. All the data points are normalized to the smallest observed RPS. The RPS of search queries follows a diurnal and day-of-week pattern. For power management, a coarse granularity epoch based latency prediction can capture this workload variability [14]. The corresponding CDF for the normalized RPS is given on the top-right sub-figure. The highest RPS rate can be up to 4 times larger than the smallest RPS. However, the results on bottom-left sub-figure shows that the Wikipedia query trace also has a high per second granularity RPS variability, in addition to the longer term variability. This suggests that it is necessary to have a per query basis design for the search engine power management. The inter-arrival time between consecutive requests also has a high uncertainty with significant variations, as shown in the bottom-right sub-figure (of Fig. 1 (b)).

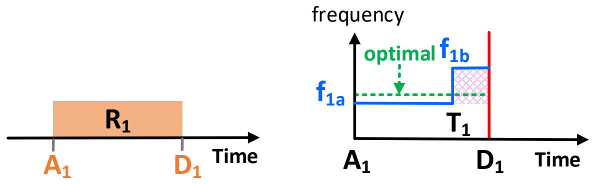
To account for the variation of incoming requests, prior works [17], [18] adopt an analytical model, which considers the queuing, for power management, upon every request arrival and departure. To estimate the per request *equivalent latency* (i.e., service time plus queuing time), they assume that each request's service time can be derived from the same distribution. Nevertheless, the results in Fig. 1 (c) invalidate this assumption. The top sub-figure presents the measured request service times on a search engine with 16 ISNs, which is deployed on our testbed platform. We find that the service times for three consecutive queries (i.e., Canada, Bobby and Tokyo) vary a lot on the same ISN server. For example, the service time of query Canada is 14 times longer than the query Tokyo on ISN-1. The CDF of service times in the bottom sub-figure confirms that this kind of variability exists across a wide range of 20K requests we measured. Although their analytical models can capture the arrival variations, it can not fully utilize the per query service time variation. Our experimental results indicate that each query's service time depends on the particular query's features and the posting list at the particular ISN, which itself can vary quite widely.

### *B. Power Management Techniques*

Energy saving for latency critical applications was specifically addressed by Lo et al. [14]. Their design, Pegasus, addresses the long-term variability of the service time, such as having a diurnal pattern, and assumes that the service time rarely changes at short timescales. Pegasus utilizes the measured deadline violation and latency headroom during the past epoch to decide on frequency settings. When the measured latency is smaller than 65% of the given time budget, the CPU frequency is reduced. The performance of Pegasus heavily relies on how quickly its feedback controller can adapt to the request variations. On the other hand, Rubik [18] observes that both the request service time and inter-arrival time have a high short-term variability. To capture the per request variability, Rubik develops an analytical model for CPU frequency selection at the granularity of every request arrival and departure. However, it only approximately estimates each request's computation demand, as every query's CPU cycles are derived from the same distribution. For latency constraints, it conservatively uses the tail of service time distribution for each request's latency prediction, thereby potentially missing many energy saving opportunities. This motivates us to have a precise per query service time prediction for improved power saving. It can be done through a multi-feature machine learning model, one which has already proved to be useful for tail latency reduction in search applications [25], [26], [36]. However, since it is impossible to develop a model with 100% accuracy without resulting in over-fitting [37], Gemini employs a two-step DVFS to meet the tight latency constraint on ISN servers.

## III. DVFS Control

To make the analysis easy to understand, we first describe our design with the assumption that there is always one request in the queue. Then, we extend it to the case of two requests, and finally to the general case of N requests.



(a) (b)

Fig. 2. Example of two-step DVFS with single request in the queue.

### A. Single Request Without Queuing

In Fig. 2 (a), request $R_1$ arrives at time $A_1$ and has to finish its work before the deadline $D_1$. Suppose that CPU is running at frequency $f_{default}$ before $R_1$ enters the queue. For power management, we predict request $R_1$'s service time with our NN model. The predicted service time $S_1^*$ is:

$$S_1^* = Predict^{NN}(Q_1, I|f_{default}) \tag{1}$$

where $Q_1$ is the request's query and $I$ is an ISN's index. Given $Q_1$ and $I$, we can obtain request $R_1$'s query features. All the predicted service times are conditioned by the default frequency $f_{default}$. Previous works [14], [18], [38] assume that a search request's service time $S$ is inversely proportional to the CPU frequency. Specifically, Chou et al. in [17] reports that a request's total work $W = Mf + C$, where $C$ is the CPU cycles and $M$ is the *memory access time*. Therefore, $W/f = M + C/f$ where $f$ is the frequency. Since we focus on the CPU management, we omit the memory access time $M$, treating it as a constant, as in prior work [17] and [18]. For simplicity, we use the service time $S$ to refer to $W/f$. Then, the above equation becomes $S = C/f$.

To validate this equation, we measured a particular search query's latency at various CPU frequencies on our platform. In Fig. 3, we see that the request's latency increases from 40ms to 97ms when the CPU frequency is slowed down from the fastest 2.7GHz to the slowest 1.2GHz frequency. Additionally, we fit a latency trending line on the same figure. Almost all the <frequency, latency> sample points are exactly on this linear trending line, confirming that a search request's processing time has a linear dependence to the CPU frequency. The time delay for the CPU to transition from one frequency to another with CPU stalls [39], is a constant, $T_{dvfs}$. If the predicted service time $S_1^*$ equals $R_1$'s actual service time $S_1$ with 100% prediction accuracy, the frequency set during the time interval $A_1$ to $D_1$ (in Fig. 2 (b)) should be constant [19]. This optimal frequency $f_1$ can be calculated as follows:

$$W_1 = W_1^* = S_1^* f_{default} \tag{2}$$

$$f_1 = W_1^*/(D_1 - A_1 - T_{dvfs}) \tag{3}$$

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

$$S_1^* = S_1 + E_1 \tag{4}$$

where $E_1$ is the prediction error. With unknown $S_1$ during the runtime, a step-wise DVFS can produce better power savings [19], [40]. To reduce the transition overhead, we limit our scheme to be a two-step design. For a two-step DVFS, shown in Fig. 2 (b), we have to solve three problems: 1.) select the initial frequency $f_{1a}$ 2.) determine the time $T_1$ for boosting and 3.) choose the boosted frequency $f_{1b}$. In Gemini, we fix the boosted frequency at maximum for the CPU core and use the predicted service time $S_1^*$ to select the $f_{1a}$. Then, $f_{1a}$ is:

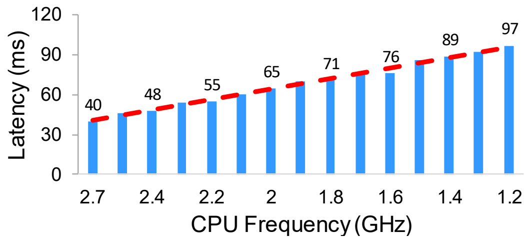


Fig. 3. The latency of search request is inversely proportional to the CPU's frequency.

$$f_{1a} = S_1^* f_{default}/(D_1 - A_1) \tag{5}$$

In the following analysis, we focus on the case when $S_1^*$ is shorter than $S_1$, as a deadline violation is more serious than energy inefficiency. If the prediction of $S_1^*$ is accurate, the line of $f_{1a}$ in Fig. 2 (b) would be straight (to $D_1$). Nevertheless, the shaded area reflects a period in which we have to boost to $f_{1b}$ to accommodate for prediction errors, so that we meet the latency requirement.

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

$$E_1^* = Predict^{Error}(Q_1, I|f_{default}) \tag{6}$$

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

$$f_{1a}(T_1 - A_1) + f_{1b}(D_1 - T_1 - T_{dvfs}) = (S_1^* + E_1^*) f_{default} \tag{7}$$

Notice that we execute the request at frequency $f_{1b}$ for a time interval $D_1 - T_1 - T_{dvfs}$, since the CPU will stall for $T_{dvfs}$ whenever we change the frequency. By combining equations 5 and 7, $T_1$ can be calculated. In the worst case, $T_1$ will be at the beginning, $A_1$, where we have to boost the frequency right away to meet the deadline. If $T_1$ is equal to $A_1$ and request $R_1$ still can not finish its work $W_1$ before the deadline $D_1$, our DVFS scheme will directly drop request $R_1$ to save energy. Dropping this kind of request will not impact the search quality seen by the client [2], as this response would be dropped by the aggregator anyway [2], [41]. The accuracy of the service time prediction is very important to the power saving achieved with our two-step DVFS. In equations 5 and 7, $f_{1a}$ will be closer to the optimal frequency $f_1$, and the boosting time $T_1$ will be closer to the deadline $D_1$, if the error of service time prediction is smaller.

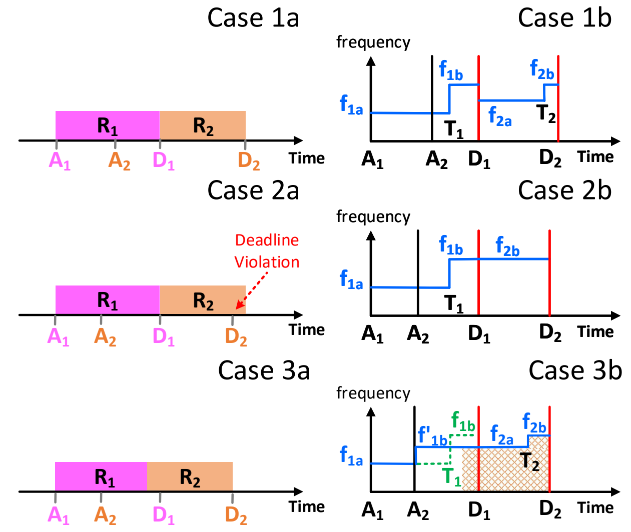


Fig. 4. Two requests in the queue. Example of critical request and non-critical request.

### B. Two Requests With Queuing

The scenario when two requests might stay in the queue is more complicated as we need to re-configure the current CPU frequency to guarantee the later request's latency requirement. Fig. 4 (*Case 1a*) depicts the arrival of the second request $R_2$ at time $A_2$ after $R_1$ started processing at time $A_1$. The deadlines for the two requests are $D_1$ and $D_2$, respectively. When the inter-arrival time, $A_2 - A_1$, is large enough as in Fig. 4 (*Case 1b*), such that request $R_2$ can finish its job within the interval $D_1$ to $D_2$, we call request $R_2$ non-critical. When a non-critical request arrives at the queue, we don't need to re-configure the current setup and request $R_2$ still uses a two-step DVFS to save power. On the other hand, request $R_2$ in Fig. 4 (*Case 2a*) will violate its deadline when the interval between $D_1$ to $D_2$ is too short. In *Case 2b*, request $R_2$ is considered to be a critical request when the following happens:

$$(D_2 - D_1)f_{2b} < (S_2^* + E_2^*)f_{default} \tag{8}$$

$W_2^{'} = (S_2^* + E_2^*)f_{default}$ is the total amount of work predicted for request $R_2$, including prediction errors. When request $R_1$ finishes, we skip step one for request $R_2$ and directly boost the CPU frequency to $f_{2b}$ (i.e., $f_{default}$). With that being the case, the maximum amount of work that can be done within the residual time $D_2 - D_1$ is $(D_2 - D_1)f_{2b}$. Request $R_2$ is critical when $(D_2 - D_1)f_{2b}$ is smaller than $W_2^{'}$. In order to guarantee $R_2$'s latency constraint in Fig. 4 (*Case 3b*), we have to boost the current frequency immediately and re-configure the second step frequency $f_{1b}$ (green line) to $f_{1b}'$ (blue line), in order to finish $R_1$ early. Then, request $R_2$ (i.e., shaded area in *Case 3b*) can begin to execute even before $D_1$. Notice that we can boost the frequency earlier only when the arrival time $A_2$ of request $R_2$ is earlier than the initial boosting time $T_1$. Otherwise, nothing can be adjusted as frequency $f_{1b}$ in *Case 3b* is already $f_{default}$, when $A_2$ is between the initial value of $T_1$ and $D_1$. To minimize the transition overhead, we select the same frequency for $f_{1b}'$ and $f_{2a}$. If a combination of $< f_{2a}, T_2, f_{2b} >$ can make request $R_2$ meet its deadline $D_2$, then request $R_1$ will definitely complete before the deadline $D_1$. So, we now focus on the calculation of $< f_{2a}, T_2, f_{2b} >$ for request $R_2$. Before selecting the frequencies and boosting time for $R_2$, we define its predicted equivalent CPU cycles $eW_2^*$ as follows:

$$eW_2^* = (S_1^* + E_1^*)f_{default} - (A_2 - A_1)f_{1a} + S_2^* f_{default} \tag{9}$$

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

$$f_{2a} = eW_2^* / (D_2 - A_2 - T_{dvfs}) \tag{10}$$

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

$$\begin{aligned} f_{2a}(T_2 - A_2 - T_{dvfs}) + f_{2b}(D_2 - T_2 - T_{dvfs}) \\ = eW_2^* + E_2^* f_{default} \end{aligned} \tag{11}$$

where we must finish the total work for the two requests before deadline $D_2$. By combining equation 10 and 11, the boosting time $T_2$ can be calculated. A special scenario of *Case 3b* is when an incoming request $R_2$ can not finish its work even if we boost the CPU frequency to $f_{default}$ immediately after it arrives. In such a case, it is safe to just directly drop request $R_2$, in the interest of saving more energy.

### C. General Case with N Requests

We now address the general case. When there are $N-1$ requests in the queue when critical request $R_N$ arrives. If request $R_N$ is non-critical, no action needs to be taken. In Fig. 5 (*Case 1a*), we give an example with $N = 3$. Request $R_1$ currently uses a two-step DVFS to save power and the next request $R_2$ is non-critical. When the critical request $R_3$ arrives at the queue in *Case 1a*, we have to boost request $R_1$'s initial frequency $f_{1a}$ to $f_{1b}'$ as in Fig. 5 (*Case 1b*). Similar to equation 9, request $R_3$'s equivalent total work $eW_3^*$ has to be calculated before making the new frequency plan. In general, request $R_N$'s equivalent total work $eW_N^*$ is:

$$\begin{aligned} eW_N^* = W_1^{residual} + \sum_{1<i<N} (S_i^* + E_i^*)f_{default} \\ + S_N^* f_{default} \end{aligned} \tag{12}$$

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

$$W_1^{residual} = (S_1^* + E_1^*)f_{default} - (A_N - A_1)f_{1a} \tag{13}$$

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

$$f_{1b}' = f_{Na} = eW_N^* / (D_N - A_N - T_{dvfs}) \tag{14}$$

In this design, all requests in between the current request $R_1$ and critical request $R_N$ adopt the same frequency to minimize

TABLE II
FEATURES FOR SERVICE TIME PREDICTION

| Query | **Time (ms)** | AMean Score | GMean Score | HMean Score | Max Score | Estimated Max Score | Score Variance | Posting List Length | # of Local Maxima |
|---|---|---|---|---|---|---|---|---|---|
| Toyota | 13 | 9.34 | 9.05 | 8.68 | 14.81 | 1131 | 5.99 | 20742 | 3084 |
| United Kingdom (*Max*) | 21 | 6.9 | 6.9 | 6.89 | 7.42 | 2144 | 0.02 | 2369024 | 2834 |

| Query | **Time (ms)** | Local Maxima above AMean | # of Max Score | Docs in 5% of Max Score | IDF | Docs in 5% of $K^{th}$ Score | Docs ever in Top-$K$ | Query Length |
|---|---|---|---|---|---|---|---|---|
| Toyota | 13 | 2639 | 1 | 199 | 6.81 | 322 | 85 | 1 |
| United Kingdom (*Max*) | 21 | 2373 | 1 | 6357 | 3.41 | 9946 | 81 | 2 |

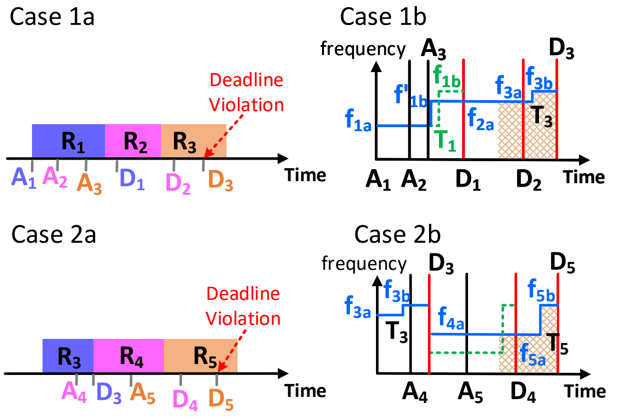


Fig. 5. Example of $N$ (=3) requests in the queue. In Case 2a, request $R_3$ and $R_4$ can finish before their deadlines if we use $f_{3a}$ as the current CPU frequency. After $R_3$ leaves, $R_5$ suffers the deadline violation if $R_4$ adopts a two-step DVFS without considering the later request $R_5$.

the frequency transition overhead. Before the critical request $R_N$ arrives, the current frequency plan has already guaranteed that the existing $N-1$ requests in the queue can meet their deadlines. Due to the arrival of critical request, we have to boost the current frequency. So, request $R_1$ to $R_{N-1}$ will finish before their deadlines. It is safe to use the same frequency for them to reduce the transition overhead. After determining $f_{Na}$ from equation 14, we use the following equation to get $T_N$, with $f_{Nb}$ to be $f_{default}$.

$$\begin{aligned} f_{Na}(T_N - A_N - T_{dvfs}) + f_{Nb}(D_N - T_N - T_{dvfs}) \\ = eW_N^* + E_N^* f_{default} \end{aligned} \quad (15)$$

Fig. 5 (*Case 2a*) shows the scenario when request $R_1$ and $R_2$ depart from the queue. Currently, request $R_3$ is still critical and new arrivals of request $R_4$ and $R_5$ are non-critical. In Fig. 5 (*Case 2b*), the non-critical request $R_4$ might adopt a two-step DVFS (green line) to save its power. However, request $R_5$ in *Case 2a* suffers a deadline violation if $R_4$ only considers its own computation demand. In Gemini with $N$ requests in the queue, we plan the CPU frequency in groups. For example, we find the next critical request ($R_5$) in *Case 2a* after the current critical request $R_3$ departures. Then, our design uses the method in *Case 1* for selecting the frequencies such that we can meet the deadlines and reduce transition overheads.

## IV. LATENCY AND ERROR PREDICTORS

For power managements, inaccurate service time estimation can result in deadline violations, when we attempt to finish the requests just-in-time [17], [18]. Gemini employs two NN models to estimate the query specific service time and prediction error, respectively. As described in the last section, the predicted service time $S^*$ is used for the initial frequency selection in our two-step DVFS, and the error predictor is for determining the boosting time $T$.

### A. *Latency Prediction*

Recent research [25], [26], [36] in the *Information Retrieval* area reports that with the adoption of selective pruning [5], [21] in search engines such as Facebook Unicorn [20] and Microsoft Bing [36], a simple linear model [42] is inadequate to get a precise per query service time prediction. A few researchers have developed sophisticated machine learning models to predict the query latency [25], [26], [36]. We considered many features in query processing and limited our selection to a few important ones. Table II shows, across the different columns, all the features used in our prediction model. We have chosen to illustrate these features with the example term query "Toyota" and phrase query "United Kingdom". The service time in column 2 is the prediction label in our model. All the other columns are prediction features. AMean score is a query term's arithmetic average score of all documents in the posting list. Similarly, the GMean score is the geometric mean score and HMean is harmonic mean. Estimated max score is an approximation of the max score based on the algorithm in [43]. In a query term's score distribution, there might exist local maxima. The "# of Local Maxima", and "Local Maxima above AMean", capture these score distribution features. Additionally, we quantify the number of documents that fall within 5% of max score and 5% of the $K^{th}$ score, where $K$ is the size of result sets. Finally, the feature "Docs ever in Top-K" is the number of documents that are fully scored by the selective pruning algorithm. For those phrase queries having more than one terms, the maximum of query terms' feature values is used.

In Gemini, we use a NN model *with only 5 hidden layers* to make the latency prediction, because it achieves a good balance between prediction accuracy and inference overhead on our platform. Each hidden layer has 128 neurons and uses the relu activation function. Our classification model is trained

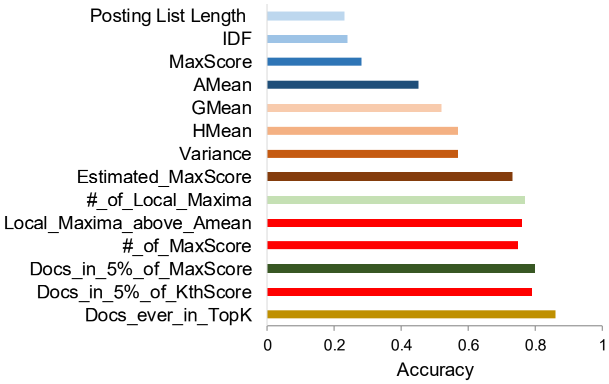


Fig. 6. The prediction accuracy when we keep adding new features from Table II. Red bars are query features that will adversely impact the prediction accuracy.

by the Adam optimization algorithm [44] with sparse categorical cross-entropy loss function. For service time scaling in Gemini DVFS, our NN model predicts each query's service time using the default CPU frequency. Next, we evaluate the feature importance on our NN model. In this experiment, we first develop a NN model with "Posting List Length" as the only feature. Then, more features are added to this NN model in the order of top to bottom as shown by the Y-axis of Fig. 6. All the features listed on the Y-axis of Fig. 6 are from Table II. In Fig. 6, we can observe that the NN model with single feature (i.e., "Posting List Length") only yields a prediction accuracy of 23%. When having more features, the prediction accuracy improves. Finally, the model with all the query features at the bottom of Fig. 6 achieves 89% prediction accuracy. Red bars in the figure, such as Local_Maxima_above_Amean, #_of_MaxScore and Docs_in_5%_of_KthScore, are query features that will degrade the prediction accuracy. In Gemini, we carefully select the features for a search engine system such that our NN model achieves the maximum prediction accuracy.

### B. Model Comparison

We now compare the prediction error and overhead results for the NN classifier, NN regressor and a simple linear classifier on the Wikipedia query trace and index. All the models are using the same query features. During the experiment, we observe that the prediction error will reduce if we train the models over more iterations. The detailed comparison results are given in Fig. 7. In Fig. 7 (a), each model's X-axis value is its prediction error and the Y-axis value is its prediction overhead. For the NN regressor with the MSE loss function, we train it by using the RMSprop optimization algorithm. Due to the selection of a MSE loss function, we define that a prediction error happens when the absolute value of true service time minus the predicted service time is larger than 4ms. On the other hand, the threshold for the (NN and linear) classifier models is 1ms as the output neuron of our classification model is at per millisecond granularity. Although the simple linear classifier has the smallest overhead of 64$\mu$s, its worst prediction error of 73% prevents us to use it. Next, the NN regressor has a better prediction error of 24% with similar prediction overhead (i.e., 66$\mu$s). However, the 24% prediction error is still too high for the search request processing with strict latency constraints. Finally, the selected NN classifier in Gemini has the smallest prediction error of only 11%. The 79$\mu$s of prediction overhead in our NN classifier is relatively small compared with a search request's total service time. In Fig. 7 (b), we plot the average request service time on both our platform and Microsoft Bing [36]. On our platform, the average request service time is around 10000$\mu$s (i.e., 10ms). Microsoft reports that the average service time for Bing search is 13470$\mu$s (i.e., 13.47ms) due to the large index [26], [45]. The average request service time is 127 times of the prediction overhead for our NN classifier.

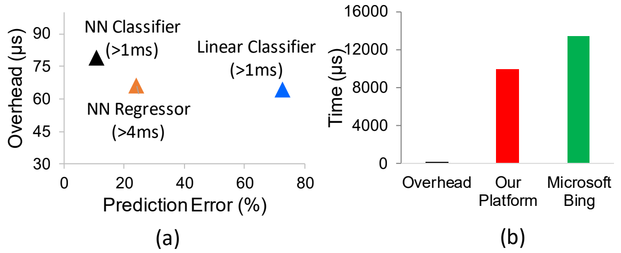


Fig. 7. (a) Prediction error and overhead for the NN models and the linear model. (b) Prediction overhead and the average request service time.

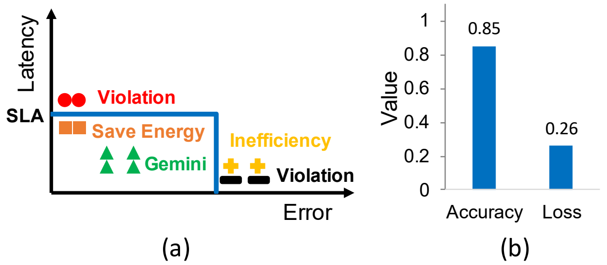


Fig. 8. (a) The impact of latency and error prediction for energy management. (b) Accuracy and loss of our error predictor.

### C. Design of an Error Predictor

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

The impact of this kind of prediction error is discussed in Fig. 8 (a). The X-axis on Fig. 8 (a) is the prediction error and the Y-axis is a request's predicted latency. If the prediction error is negligible, a long request (red circle) may violate the SLA requirement while a short request (orange square) may be slowed down for power saving and still not violate the SLA. However, even for such a short request, if the prediction error is negative and large, this will result in the predicted service time $f(X_i)$ being much shorter than $y_i$, causing us to select a much lower CPU frequency than appropriate and result in a violation of the SLA. Similarly, large positive prediction errors will result in energy inefficiency because we will select too high a frequency. In order to solve this problem, Gemini uses an additional predictor to estimate a request's service time prediction error [26]. For the predicting error, we use the same query features as listed in Table II. To train another NN model for error prediction, the label $E = f(X_i) - y_i$ can be easily obtained in training set since we can keep track of the measured request latencies in the past. Fig. 8 (b) shows the prediction accuracy for errors is 85%.

## V. GEMINI IMPLEMENTATION

Gemini is implemented on the Solr search engine. The high level architecture of Solr search engine is shown in Fig. 9. When a client's search request arrives, a SearchHandler will forward the request to the IndexSearcher, which calls the Apache Lucene APIs for retrieving the relevant documents. The Solr search engine is written in Java. To implement Gemini, we wrap the IndexSearcher and Lucene Index Searching in Fig. 9 as a Java Callable task. The reason for doing this is that the Java Executor framework can automatically handle Callable tasks in a Blocking Queue and provide mechanisms for thread management. When a search request arrives, we submit its task to the Blocking Queue and wait for an idle working thread to process its query. Currently, our implementation only has one working thread as we focus on single core power management. With multiple CPU cores, we can maintain a separate queue for each core and have a global broker to distribute the incoming requests to each core, as suggested in [25]. Each core will manage its power consumption independently by using Gemini's DVFS scheme. This is work we plan for the near future. Finally, we conduct the ISN power management upon the arrival and departure of every task.

We use the TensorFlow [37] to achieve Gemini's NN prediction model. The inference time for prediction is only 73 - 83ms, compared with tens of milliseconds for the query's service time. As Gemini is implemented as a part of the ISN, we need user-space DVFS control. Gemini leverages the Advanced Configuration and Power Interface to update the CPU core's frequency at runtime. Our frequency enforcement is achieved by manipulating each core's "scaling_setspeed" file. When this device file is changed, Linux triggers a group of system calls that take only 40$\mu$s totally to update the CPU core's frequency. The extra latency overheads from the Gemini implementation are negligible.

Our experimental setup has two machines connected by a 1G Ethernet link: one as the client and the other as the search engine server. The server machine has an Intel Xeon E5-2697 CPU with 24 cores, 128G memory running CentOS 7 operating system. Three representative query traces are used in our experiments: the Wikipedia [35], the Lucene nightly benchmark [16] and the TREC Million Query Track (MQT) [8] traces. On the server side, we deploy a Solr search engine with 12 ISNs. In the search engine, we index the complete dump of entire English Wikipedia web pages on December 1st, 2018. This 65GB index has a total of 34 millions documents. The search engine machine supports per core frequency scaling. The CPU frequency can be selected in the range of 1.2 GHz to 2.7 GHz. The default CPU frequency is 2.7 GHz. For power measurement, our CPU has sensors to monitor per socket's energy consumption. The accumulated energy consumption of a CPU socket is stored in a Machine Specific Register (MSR). By writing an energy measurement daemon which reads the MSR register every 1 second through the Running Average Power Limit (RAPL) interface, we can obtain a CPU socket's power consumption. As the measured CPU energy per second is the socket package energy, we deploy 12 single-working-thread ISNs on the 12-cores CPU chip. Then, each ISN is bound to one core of the CPU chip. The 12 ISNs receive the same search queries from our Solr aggregator but schedule their core's frequency independently.

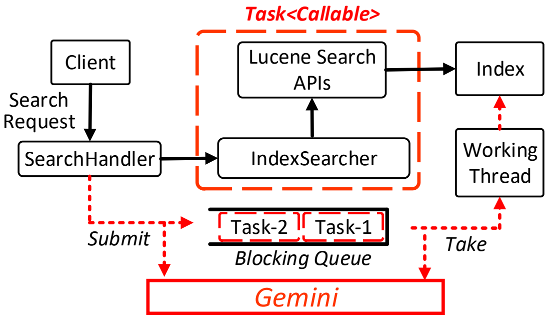


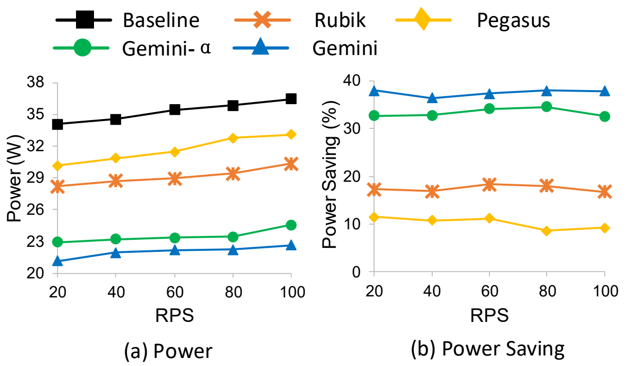
Fig. 9. Gemini implementation overview.

## VI. EVALUATION

Gemini is first evaluated with a range of server loads, in terms of RPS, to show the significant CPU power saving while still achieving the latency constraints. We also perform trace-driven experiments using three different query traces to characterize Gemini's power management under realistic workloads. Finally, we disable each component of Gemini in turn, to show the underlying reasons for the increased power saving of Gemini.

### A. Power Saving

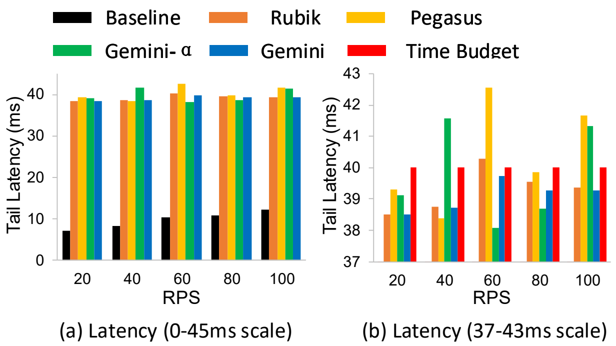
In Fig. 10, the baseline doesn't have any power management and always uses the default 2.7GHz CPU frequency. Rubik [18] is an analytical power management framework which adjusts the CPU frequency on every request arrival and departure according to the tail (95th percentile) of the service time distribution. Then, we compare with Pegasus [14], which is a feedback based power management scheme. It measures the request's latency periodically and selects the highest CPU frequency if a deadline violation happens. To match the search load we use over 1000s, we scale Pegasus' 5s epoch length (over a 12-hours period for the load) to 125ms in our experiment, so as to have the same ratio between epoch length and load length. Apart from the complete design of



(a) Power

(b) Power Saving

Fig. 10. The CPU power results for various RPSs. Time budget uses the tail latency at high load.



(a) Latency (0-45ms scale)

(b) Latency (37-43ms scale)

Fig. 11. Tail latency results for different RPS. Part (b) is the tail latency with 37-43ms scale for part (a).

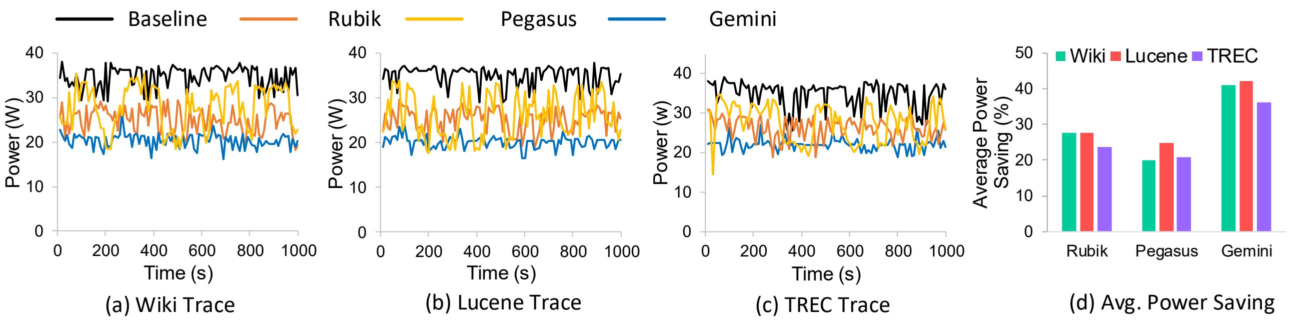
Gemini, we also implement another version of our scheme called Gemini-$\alpha$, in which we don't use the second NN model for error prediction. Instead, we use the moving average of errors in the past 60 request arrivals to estimate the current request's latency prediction error. By doing this, we quantify the benefit of having the second predictor for the error in our design on CPU power saving and request tail latency.

To compare across the different power management frameworks, we use the same query workload (i.e., Wikipedia trace) on the Solr search engine. The results are shown in Fig. 10. Each request rate (in RPS) is maintained for 120s to obtain the corresponding average CPU power consumed at that server load. Fig. 10 (a) shows the CPU powers for RPS varying from 20 to 100. In our experiment, the measured CPU power includes the power consumed by CPU cores, caches and other components on the chip. The results show that CPU power increases for all the frameworks, as we increase the request intensity. For example, the CPU power for the baseline case increases from 34W to 36.5W when the RPS value goes up from 20 to 100. Higher server load results in more request queuing. Thus, the latency slack that power management frameworks used to slow down requests is reduced. Among all the power management alternatives, Pegasus consumes the highest CPU power, because of its epoch based design. Rubik performs better than Pegasus, but still consumes more CPU power than our Gemini variants. The results in Fig. 10 (a) shows that the complete design of Gemini uses the least CPU power and outperforms the Gemini-$\alpha$ which doesn't include a second error predictor. The reason is that the inaccurate moving average prediction in Gemini-$\alpha$ makes our two-step DVFS enter the second frequency step too early in its goal of meeting the latency constraint. So, Gemini-$\alpha$ consumes more CPU power than Gemini across the entire range of loads. The second error predictor in Gemini is clearly beneficial in reducing the CPU power consumption.

Fig. 10 (b) presents the power saving in percentage compared to the baseline. Although the actual power consumption increases linearly in Fig. 10 (a) for each technique, the percentage power saving remains similar for each technique compared to the baseline. As the relative differences across frameworks are similar for the range of RPS, we focus on the high server load of 100 RPS, where power saving is more challenging to achieve. At this high server load, Pegasus saves 9.23% of CPU power while Rubik achieves 16.8% power saving compared to the baseline. The power saving for Gemini-$\alpha$ is 32.7%, which is 1.95 times better than Rubik. The complete design of Gemini performs the best, with 37.9% CPU power saving compared to the baseline. This is 2.25 times better than Rubik, and is even better when compared to Pegasus.

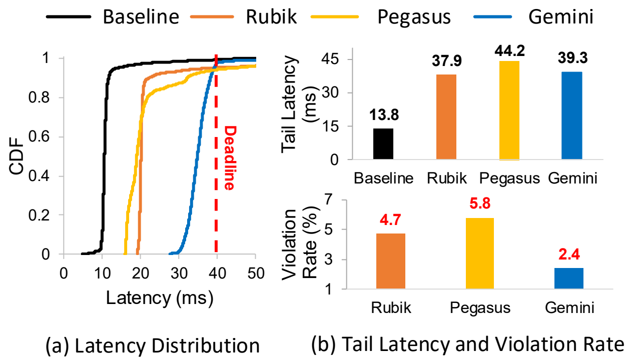
## B. *Tail Latency*

For the same experiment as above, the 95th percentile tail latency for different RPS is shown in Fig. 11. In Fig. 11 (a), we observe that the tail latency for the baseline, with a fixed 2.7 GHz frequency, increases with load, due to the request queuing. For example, the request's tail latency is 12.3ms at 100 RPS, compared with the 7.05ms at 20 RPS. To utilize the latency slack, and save CPU power, power management frameworks such as Rubik and our Gemini will slow down the request's processing time to be close to the 40 ms latency bound. Fig. 11 (a) shows that all the compared frameworks can roughly achieve this performance constraint. In Fig. 11 (b), we look into the tail latency at a finer granularity (i.e., the 37-43ms range), with the red bar in the figure being the time budget given. At 100 RPS, the tail latency of Rubik is 39.36ms and the others are similar, across the range of RPS (all within about a 1ms range). Because Pegasus has a feedback based controller to select the CPU frequency, it has a higher tail latency variations across the range of loads. Pegasus violates the 95th tail latency constraint by achieving 41.67ms at 100 RPS. Due to the inaccurate error estimation in Gemini-$\alpha$, which uses a simple moving average from recently processed queries, Gemini-$\alpha$ sometime has a higher 95th tail latency than the target time budget of 40 ms. Finally, the complete design of Gemini along with the error predictor consistently meets the 40ms tail latency constraint all the time. Compared with Rubik and Pegasus, Gemini meets the deadline requirement but still achieves higher CPU power savings, as shown in Fig. 10 and 11. This is because we accurately "reshape" the request's latency distribution, so that more of the requests have their latencies closer to the time budget.



(a) Wiki Trace (b) Lucene Trace (c) TREC Trace (d) Avg. Power Saving

Fig. 12. Power consumption results for the Wikipedia, Lucene and TREC traces.



(a) Latency Distribution (b) Tail Latency and Violation Rate

Fig. 13. Gemini achieves the smallest tail latency and lowest deadline violation rate.

### C. Trace-Driven Characterization

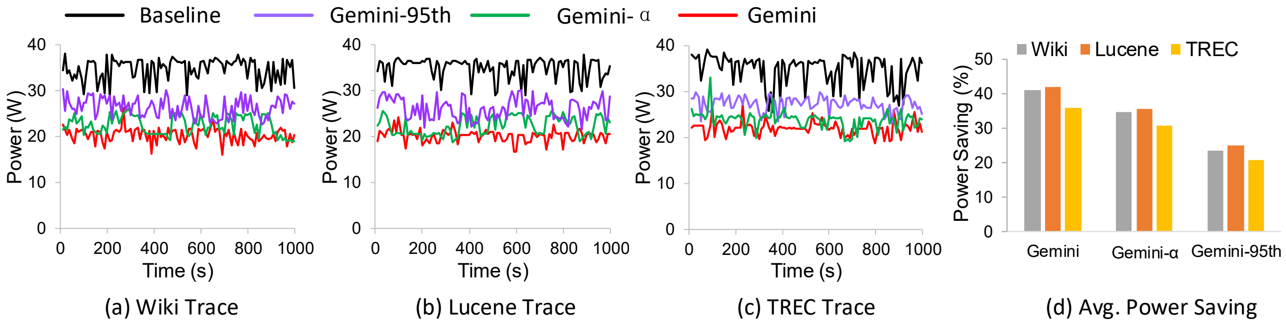
For the next experiment, we evaluate Gemini with a few trace-driven workloads, for sensitivity analysis. On the 1000s traces, Gemini is compared with Rubik and Pegasus. Fig. 12 (a) and (b) present the CPU power for the Wikipedia and Lucene nightly benchmark traces, respectively. Similar to prior works [4], [8], [46], we also show the comparison results for the TREC query traces (widely used in the information retrieval area), in Fig. 12 (c). For these three query traces, we still use 40ms as the target tail latency. With all the traces, power consumptions of the baseline are in the range of 29.1W to 38.2W as the server load varies over time. Pegasus still consumes the most CPU power among the alternative frameworks, primarily because of its epoch based design. Rubik outperforms Pegasus at most of the time. On the other hand, Gemini leverages a two-step DVFS equipped with a NN latency predictor for the initial frequency selection and a second predictor for the boosting time determination to improve the CPU power saving. The results in Fig. 12 (a), (b) and (c) show that Gemini consumes the least amount of CPU power, across all the three traces. We show the average power savings in Fig. 12 (d). Rubik achieves around 23.7%-27.8% CPU power savings with the three traces. The average power savings for Pegasus on the three traces are in the range of 20.07% to 24.72%. The best power management among these schemes, Gemini, achieves up to 42.15% power saving on the Lucene trace which is 1.53 times better than Rubik and outperforms Pegasus by 70.5%.

In addition to the power saving, the request latency distribution for each scheme on the Wikipedia trace is shown in Fig. 13 (a). Rubik, Pegasus and Gemini save CPU power because they can shift the "knee" of latency distribution to the right, towards the deadline. Although the tails of their latencies are similar in Fig. 13 (a), Gemini is able to slow down more requests, such that they are closer to the deadline. This allows Gemini to have better energy savings. The detailed tail latency and deadline violation rate for this trace-driven experiment are presented in Fig. 13 (b). On the top sub-figure, the baseline has the smallest 95th tail latency with 13.8ms (much smaller compared to our 40ms time budget). Rubik utilizes this latency slack to shift the tail to 37.9ms, completing just before the deadline, to save energy. The 95th tail latency on Gemini is 39.3ms. However, Pegasus has a 95th tail latency of 44.2ms, thereby resulting in more than 5.8% of the requests having deadline violations as seen in the bottom sub-figure. Rubik always uses the 95th percentile on the service time distribution for frequency selection. Thus, Rubik's deadline violation rate is lower, at 4.7%. But, by having a precise per query service time prediction and using a second error predictor to reduce the deadline violation, Gemini achieves a 2.4% deadline violation rate which is less than half of that for Pegasus. Thus, Gemini achieves a much better balance between the CPU power saving and deadline violation rate than the alternatives we compare.

### D. Breakdown of Power Saving

In Gemini, our two-step DVFS initially selects a low CPU frequency to save power, but incorporates mechanisms to make sure that the initial selected frequency is not too low to miss the deadline. Then, our two-step DVFS boosts the initial frequency at just the right time, such that the deadline is met. In fact, PACE [19] theoretically proves that a step-wise DVFS scheme produces the optimal CPU power saving when a task's total work is unknown. Both synthetic and trace-driven experiments show a significant power saving for Gemini. To further enhance the power management, we develop two separate latency and error predictors with high accuracy. The latency predictor enables a suitable initial frequency selection in our two-step DVFS and the error predictor enables our scheme to boost the initial frequency at the correct time.

In this experiment, we examine the reasons behind the significant power saving of Gemini on three different query traces. We develop two variants of Gemini. In particular, we



(a) Wiki Trace (b) Lucene Trace (c) TREC Trace (d) Avg. Power Saving

Fig. 14. Comparing Gemini, Gemini-$\alpha$, and Gemini-95th.

seek to determine the impact of our NN predictors. First, Gemini-$\alpha$ described in the previous experiments is reused. Gemini-$\alpha$ doesn't use the second error predictor of Gemini at all. Building on Gemini-$\alpha$, Gemini-95th also removes its latency predictor. Instead, similar to Rubik, we use the 95th percentile of service time distribution to estimate a query's latency in Gemini-95th. In Fig. 14 (a), (b) and (c), we report the CPU power variation with time for the Wikipedia, Lucene and TREC traces. For all the traces, the power consumption of Gemini-$\alpha$ is slightly higher than Gemini. This is because the moving average in Gemini-$\alpha$ is unable to provide a measure of each request's precise residual work. Thus, the two-step DVFS has to boost the CPU frequency earlier to achieve a lower deadline violation rate. When we further disable the latency predictor and use the 95th percentile of the service time distribution for frequency selection, Gemini-95th consumes even more CPU power than both Gemini-$\alpha$ and Gemini. The average power saving results are shown in Fig. 14 (d). When comparing Gemini with Gemini-$\alpha$, we find that the use of a second error predictor brings us around 6.53% more power saving on the Lucene trace. On the other hand, the power saving improvement contributed by our service time predictor in Gemini is 10.8% on the Lucene trace. On the TREC trace, the average power saving of Gemini is 36.09%. Breaking this down, when only the two-step DVFS (without the predictors) is used, as in Gemini-95th, the power saving is 58% of Gemini. By having the latency predictor but not the error predictor in Gemini-$\alpha$, we achieve 86% of the complete design's power saving. Thus, we conclude that it is important to have both the two-step DVFS *and* the two predictors in Gemini for significant power savings.

## VII. Related Work

To improve the energy efficiency of data centers, most of the existing work on server power management is based on DVFS or Sleep states techniques. PowerNap [9] dynamically switches the server state between a minimal power consumption "nap" state and a high performance active state, to accommodate workload variations. Based on PowerNap, DreamWeaver [11] coalesces requests across multiple cores so that some cores can enter deeper sleep states. Instead of using request coalescing, DynSleep [47] procrastinates the processing of requests at a single core while still avoiding deadline violations. By doing this, the CPU core can enter the deepest sleep state to save more energy. KnightShift [12] designs a heterogeneous server architecture by having a low power consumption "Knight" node. When the server utilization is low, KnightShift puts the entire server into sleep, except for the Knight node. The low utilization workloads are served by the Knight node when other server components go to sleep. On the other hand, Pegasus [14] proposes a feedback based DVFS scheme to save server power. Similarly, TimeTrader [15] considers both the network slack and server slack to save power for latency-critical applications. Rubik [18] is a fine grain DVFS scheme which leverages a service time distribution to select frequency for critical requests. Both Rubik and Gemini use an analytical model to capture the request arrival variation, but our framework also adopts two NN models to fully utilize the per query service time variation.

## VIII. Conclusion

It is difficult to predict a search request's compute demand perfectly. This becomes a big challenge for managing power consumption of search engines that have to process latency-critical queries. To save power and meet strict latency constraints, we presented Gemini that employs a two-step DVFS policy for CPU frequency control. Our two-step DVFS utilizes a request's service time to initially slow down the query processing. To achieve good power savings, a precise NN based latency predictor is designed in Gemini. In the second step of our DVFS scheme, we have to properly boost the CPU frequency to meet a request's deadline. A separate error predictor is developed in Gemini to determine the correct time for frequency boosting. We have shown that the overhead due to NN based predictors is negligible. After implementing Gemini in the well known Solr search engine, experimental results on three representative query traces show that our framework can achieve up to 41% CPU power saving and outperform other state-of-the-art schemes, while reducing the deadline violation rates.

## Acknowledgment

We thank the US NSF for their generous support through grants CCF-1815643 and CNS-1763929.

# References

[1] X. Bai, I. Arapakis, B. B. Cambazoglu, and A. Freire, "Understanding and leveraging the impact of response latency on user behaviour in web search," *ACM Trans. Inf. Syst.*, vol. 36, no. 2, Aug. 2017.

[2] J.-M. Yun, Y. He, S. Elnikety, and S. Ren, "Optimal aggregation policy for reducing tail latency of web search," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '15. Association for Computing Machinery, 2015, p. 63–72.

[3] Y. Kim, J. Callan, J. S. Culpepper, and A. Moffat, "Load-balancing in distributed selective search," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '16. Association for Computing Machinery, 2016, p. 905–908.

[4] A. Kulkarni, A. S. Tigelaar, D. Hiemstra, and J. Callan, "Shard ranking and cutoff estimation for topically partitioned collections," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM '12. Association for Computing Machinery, 2012, p. 555–564.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient query evaluation using a two-level retrieval process," in *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, ser. CIKM '03. Association for Computing Machinery, 2003, p. 426–434.

[6] N. Tonellotto, C. Macdonald, and I. Ounis, "Efficient and effective retrieval using selective pruning," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, ser. WSDM '13. Association for Computing Machinery, 2013, p. 63–72.

[7] R. Aly, D. Hiemstra, and T. Demeester, "Taily: Shard selection using the tail of score distributions," in *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '13. Association for Computing Machinery, 2013, p. 673–682.

[8] H. R. Mohammad, K. Xu, J. Callan, and J. S. Culpepper, "Dynamic shard cutoff prediction for selective search," in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, ser. SIGIR '18. Association for Computing Machinery, 2018, p. 85–94.

[9] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. Association for Computing Machinery, 2009, p. 205–216.

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

[11] D. Meisner and T. F. Wenisch, "Dreamweaver: Architectural support for deep sleep," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. Association for Computing Machinery, 2012, p. 313–324.

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

[13] Y. Liu, S. C. Draper, and N. S. Kim, "Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. IEEE Press, 2014, p. 313–324.

[14] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. IEEE Press, 2014, p. 301–312.

[15] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "Timetrader: Exploiting latency tail to save datacenter energy for online search," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. Association for Computing Machinery, 2015, p. 585–597.

[16] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. Association for Computing Machinery, 2017, p. 625–638.

[17] C. Chou, L. N. Bhuyan, and D. Wong, "$\mu$dpm: Dynamic power management for the microsecond era," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 120–132.

[18] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. Association for Computing Machinery, 2015, p. 598–610.

[19] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," in *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '01. Association for Computing Machinery, 2001, p. 50–61.

[20] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang, "Unicorn: A system for searching the social graph," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1150–1161, Aug. 2013.

[21] S. Ding and T. Suel, "Faster top-k document retrieval using blockmax indexes," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '11. Association for Computing Machinery, 2011, p. 993–1002.

[22] R. Baeza-Yates, A. Gionis, F. P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "Design trade-offs for search engine caching," *ACM Trans. Web*, vol. 2, no. 4, Oct. 2008.

[23] R. Fagin, "Combining fuzzy information: An overview," *SIGMOD Rec.*, vol. 31, no. 2, p. 109–118, Jun. 2002.

[24] V. N. Anh and A. Moffat, "Pruned query evaluation using pre-computed impacts," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '06. Association for Computing Machinery, 2006, p. 372–379.

[25] C. Macdonald, N. Tonellotto, and I. Ounis, "Learning to predict response times for online query scheduling," in *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '12. Association for Computing Machinery, 2012, p. 621–630.

[26] S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi, "Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, ser. WSDM '15. Association for Computing Machinery, 2015, p. 7–16.

[27] "Apache solr," https://lucene.apache.org/solr/.

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

[29] A. Kulkarni and J. Callan, "Selective search: Efficient and effective search of large textual collections," *ACM Trans. Inf. Syst.*, vol. 33, no. 4, Apr. 2015.

[30] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay, "Accurately interpreting clickthrough data as implicit feedback," *SIGIR Forum*, vol. 51, no. 1, p. 4–11, Aug. 2017.

[31] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Trans. Database Syst.*, vol. 23, no. 4, p. 453–490, Dec. 1998.

[32] G. E. Pibiri, M. Petri, and A. Moffat, "Fast dictionary-based compression for inverted indexes," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, ser. WSDM '19. Association for Computing Machinery, 2019, p. 6–14.

[33] M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck, "Accelerated query processing via similarity score prediction," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR'19. Association for Computing Machinery, 2019, p. 485–494.

[34] H. Wang, S. Kim, E. McCord-Snook, Q. Wu, and H. Wang, "Variance reduction in gradient exploration for online learning to rank," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR'19. Association for Computing Machinery, 2019, p. 835–844.

[35] G. Urdaneta, G. Pierre, and M. van Steen, “Wikipedia workload analysis for decentralized hosting,” *Comput. Netw.*, vol. 53, no. 11, p. 1830–1845, Jul. 2009.

[36] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, “Predictive parallelization: Taming tail latencies in web search,” in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, ser. SIGIR ’14. Association for Computing Machinery, 2014, p. 253–262.

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16. USENIX Association, 2016, p. 265–283.

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

[39] M. Alian, A. H. M. O. Abulila, L. Jindal, D. Kim, and N. S. Kim, “Ncap: Network-driven, packet context-aware power management for client-server architecture,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 25–36.

[40] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark, “Formal online methods for voltage/frequency control in multiple clock domain microprocessors,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. Association for Computing Machinery, 2004, p. 248–259.

[41] B. B. Cambazoglu, V. Plachouras, and R. Baeza-Yates, “Quantifying performance and quality gains in distributed web search engines,” in *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR ’09. Association for Computing Machinery, 2009, p. 411–418.

[42] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates, “A pipelined architecture for distributed text query evaluation,” *Inf. Retr.*, vol. 10, no. 3, pp. 205–231, Jun. 2007.

[43] C. Macdonald, I. Ounis, and N. Tonellotto, “Upper-bound approximations for dynamic pruning,” *ACM Trans. Inf. Syst.*, vol. 29, no. 4, Dec. 2011.

[44] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *International Conference on Learning Representations (ICLR)*, 2015.

[45] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt, “Early exit optimizations for additive machine learned ranking systems,” in *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, ser. WSDM ’10. Association for Computing Machinery, 2010, p. 411–420.

[46] Z. Dai, C. Xiong, and J. Callan, “Query-biased partitioning for selective search,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM ’16. Association for Computing Machinery, 2016, p. 1119–1128.

[47] C.-H. Chou, D. Wong, and L. N. Bhuyan, “Dynsleep: Fine-grained power management for a latency-critical data center application,” in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED ’16. Association for Computing Machinery, 2016, p. 212–217.