Swan: A Two-Step Power Management for Distributed Search Engines

Liang Zhou
University of California, Riverside
lzhou008@ucr.edu

Laxmi N. Bhuyan
University of California, Riverside
bhuyan@cs.ucr.edu

K. K. Ramakrishnan
University of California, Riverside
kk@cs.ucr.edu

ABSTRACT

The service quality of web search depends considerably on the request tail latency from Index Serving Nodes (ISNs), prompting data centers to operate them at low utilization and wasting server power. ISNs can be made more energy efficient utilizing Dynamic Voltage and Frequency Scaling (DVFS) or sleep states techniques to take advantage of slack in latency of search queries. However, state-of-the-art frameworks use a single distribution to predict a request's service time and select a high percentile tail latency to derive the CPU's frequency or sleep states. Unfortunately, this misses plenty of energy saving opportunities. In this paper, we develop a simple linear regression predictor to estimate each individual search request's service time, based on the length of the request's posting list. To use this prediction for power management, the major challenge lies in reducing miss rates for deadlines due to prediction errors, while improving energy efficiency. We present Swan, a two-**S**tep po**W**er m**A**nagement for distributed search e**N**gines. For each request, Swan selects an initial, lower frequency to optimize power, and then appropriately boosts the CPU frequency just at the right time to meet the deadline. Additionally, we re-configure the time instant for boosting frequency, when a critical request arrives and avoid deadline violations. Swan is implemented on the widely-used Solr search engine and evaluated with two representative, large query traces. Evaluations show Swan outperforms state-of-the-art approaches, saving at least 39% CPU power on average.

CCS CONCEPTS

• **Hardware** → **Enterprise level and data centers power issues**; **Chip-level power issues**.

KEYWORDS

Power management, tail latency, search engine, DVFS, frequency boosting

ACM Reference Format:
Liang Zhou, Laxmi N. Bhuyan, and K. K. Ramakrishnan. 2020. Swan: A Two-Step Power Management for Distributed Search Engines. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '20), August 10–12, 2020, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3370748.3406573

1 INTRODUCTION

Latency-sensitive applications such as Web Search are critical to a data center operator's revenue. The service quality of web search is significantly affected by the request processing latency at Index Serving Nodes (ISNs) [8], which store the document index and retrieve matching results for a query. To meet the strict tail latency constraints, ISN servers typically run at low utilization [10]. Thus, there exists a latency slack between a query's latency and its deadline for most search requests [14]. But, low server utilization of ISNs wastes lots of energy [9, 16].

Energy inefficiency of search applications has prompted plenty of prior research [2, 5, 6, 14, 17] to exploit the latency slack by slowing down request processing and finishing the job just-in-time. Most of the research has been based on two techniques: Dynamic Voltage and Frequency Scaling (DVFS) and Sleep States. DVFS schemes such as Rubik [5] and Pegasus [6] dynamically adjust the CPU's frequency, while guaranteeing that the high percentile latency constraint is not exceeded. Sleep states frameworks such as PowerNap [9] and DreamWeaver [10] put the server into sleep during idle periods. Another paper has considered a combination of sleep states and DVFS to meet the deadline challenge [2].

State-of-the-art power managements for search engines seek to estimate each request's service time and then properly re-configure the current power management setup. A common approach for service time prediction [2, 5, 6, 10, 14] is to assume that each request's CPU cycles can be estimated from the same single distribution, based on offline profiling. By using the high percentile tail of this service time distribution and then deriving the CPU frequency or sleep states, they seek to achieve a low deadline violation rate while also slowing down search queries. However, the drawback of this approach is that individual queries have different distributions giving rise to inaccurate service time prediction. The major challenge here is that the service time prediction cannot be 100% accurate, which may result in energy inefficiency (because of longer predicted service times) or worse, even deadline violations (due to shorter predicted times).

An intuitive solution to overcome this energy wastage is to develop an individual query-specific service time predictor. An ISN server scores each document on the search query's posting list [1, 8] one by one to find the most relevant results for a search query. Each query's posting list is different and stored at the ISN server during the offline indexing phase. This information retrieval workflow motivates us to develop a simple Linear Regression model with negligible overheads to predict each search request's service time. This service time is *roughly* proportional to the corresponding posting list length. The posting list length is the existing term statistic of the search index [8]. Even then, we realize that the service time prediction is not accurate and some sort of correction process needs to be applied.

To overcome the above challenge, we present Swan, a two-**S**tep po**W**er m**A**nagement for distributed search e**N**gines. Our major goal is to properly schedule the various gating domains in sleep states or clock domains in DVFS [15] for optimal power saving achieving a very limited deadline violation rate. We focus on the DVFS technique in this paper, but our framework also applies to sleep states based schemes. Upon every request arrival or departure, the first step in Swan selects an initial CPU frequency f_a according to the predicted service time S^*. The first step makes sure that the power saving is optimal based on current estimation of the service time. But this prediction is likely to have some error, E. Then, the second step in Swan judiciously boosts the CPU frequency to f_b at the *correct* time T, to catch up with the request's deadline. Determining T accurately is crucial to the success of a two-step DVFS strategy. In our design, T is chosen based on a detailed analysis of the latency-constrained power saving. Additionally, we adjust the original boosting time T whenever a critical request arrives at the queue, because the original boosting time T might cause later requests to violate their deadlines.

Our paper has some resemblance to the step-wise DVFS scheme, proposed in [7]. However, the theoretical framework is difficult to implement in practice. To validate our design, Swan is implemented in the commercial Solr search engine, which is deployed on a server with user-space DVFS control and power management. Experimental results on Wikipedia [13] and the Lucene nightly benchmark [4] query traces prove that Swan outperforms both Rubik [5] and Pegasus [6], achieving at least 39% CPU power saving on average. Our major contributions in this paper include:

- We first develop a simple linear regression model with negligible overheads to predict each search request's service time, at runtime depending on the posting list.
- To minimize the impact of prediction errors, we propose a two-step DVFS scheme which properly boosts the CPU frequency to catch-up to deadlines, on a per query basis.
- Third, we consider Swan under the general case where there are multiple request arrivals, and reconfigure the original boosting time for the CPU to consider the new request, thus avoiding deadline violations.
- Finally, Swan is implemented on a real platform using the commonly used Solr search engine in production, and energy saving is evaluated with two representative query traces.

2 BACKGROUND AND MOTIVATION

Search Engine Architecture. Fig. 1 (a) depicts the partition aggregation architecture of a distributed search engine [12]. The database utilized by a search engine typically contains billions of documents which are partitioned among a group of ISN servers [11]. The front-end (e.g., a web server) forwards clients' search requests to an aggregator. The aggregator generates internal parallel queries to all the ISN servers to retrieve matched documents. To meet user-perceived latency requirements, the ISNs are required to complete the task for a query within a given time budget (i.e., deadline), so that the aggregator can collect all the ISN's results and report the final result to the client.

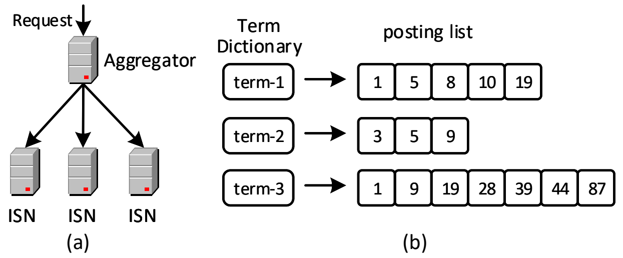


Figure 1: (a) The partition-aggregate architecture of search engine. (b) Inverted index format.

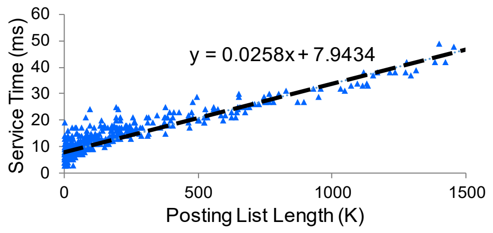


Figure 2: Search request's service time is roughly linear to its query's posting list length.

Fig. 1 (b) shows the format of an inverted index [1] stored on each ISN server. A search query phrase typically consists of a few query terms. Whenever an ISN receives a search request from the aggregator, it looks up the term dictionary for each query term in the local index and creates a corresponding posting list [8]. Then, each document on the posting list is scored one by one. This search workflow suggests that *a search request's service time is roughly proportional to the number of postings (i.e., document) for a search query [1, 8]*. As shown in Fig. 1 (b), each query term may have a different posting list length, thus resulting in a variable service time for each search request. To validate this hypothesis, we measure request service times on our platform for different posting list length, shown in Fig. 2. On the figure, we also plot a trending line for a least squares linear regression model. The trending line confirms that a search request's service time is roughly linear to its posting list length, subject to some prediction errors.

Motivation. In order to make the ISN server energy efficient, earlier power management schemes use DVFS [5, 6] or Sleep States [2, 10] and exploit the latency slack to slow down a request's processing so as to finish the retrieval of a document just-in-time. Fig. 3 plots the actual service time distribution obtained from our platform. Typically, we specify the request deadline as a high percentile tail latency under high load [2]. However, ISN servers in data centers are usually operated at low utilization levels [10] and requests may finish well before their deadlines. To take advantage of the latency slack, prior works [2, 5, 10] assume that a search request's service time S follows the same distribution $P[S = c]$. Then, they leverage the slack between the 95^{th} percentile of the latency distribution $P[S = c]$ and the deadline to save power while guaranteeing a deadline violation rate less than 5%. However, a query's actual service time might be much shorter than the distribution's tail and promises a significant energy saving improvement, if we use the query specific service time to slow down request processing.

Fig. 2 shows that a search request's service time is roughly linear to its posting list length. This motivates us to develop a simple

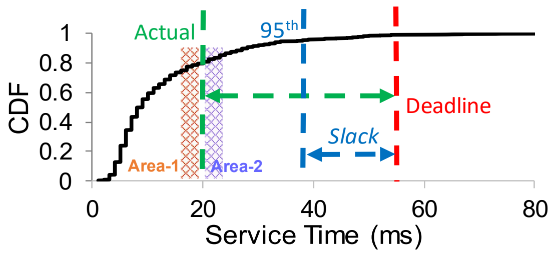


Figure 3: Predicted service time promises significant energy saving improvements, but we needs a two-step DVFS control to guarantee the latency constraints.

linear regression model to predict each request's service time during runtime, with negligible overheads. Each query's posting list length is already stored in the ISN index as a term statistic [8]. The major challenge of course is that it is impossible to have a model with 100% prediction accuracy. If the predicted service time is longer than its actual value, we waste some CPU energy (i.e., area-2). What is worse is that deadline violations will happen when the predicted service time is shorter than the actual value (i.e., area-1). To meet this challenge, we propose a novel two-step DVFS scheme. Our design, Swan, first selects an initial frequency for power optimization and then boosts it at the proper time to meet latency constraints.

Comparison with Prior Works. A few researchers have theoretically shown that a step-wise DVFS can produce better energy savings even though the actual service time is unknown [7, 15]. While the two previous papers dealt with minimizing the energy consumption of a single request, the work in [4] addresses multiple arrivals based on adjustments at a time epoch. In this paper, we consider this problem in a practical environment with a real application and using an implementation on a testbed rather than the theoretical simulation. To limit complexity, we restrict our power management approach for search engines to have a two-step scheme for reducing the transition time and power [2]. One major difference in Swan is that we use the runtime query phrase and its posting list length to predict the request's service time and then calculate the CPU frequency and boosting time. However, prior works [4, 7] still utilize sampling of the distributions to estimate the residual task time and then select the next step in the CPU frequency. The profiled distributions have to be obtained offline as a prior knowledge. Finally, another important feature of some prior research [3, 4, 15] is that they need feedback signals to make the control decision, with the same criteria to enter the next step of frequency. Swan has a query specific design with an initial frequency and a boosting time depending on the predicted service time and queuing conditions, without the overhead and delay for feedback.

3 DESIGN

We now provide details of Swan. 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 general case with N requests.

Single Request. As shown in Fig. 4 *Case 1a*, search request R_1 arrives at time A_1 and has to finish its work before its deadline D_1. Suppose the CPU is running at the constant default frequency f_d before R_1 enters the queue. For power management, we predict request R_1's service time with our Linear Regression (LR) prediction model. The predicted service time S_1^* is:

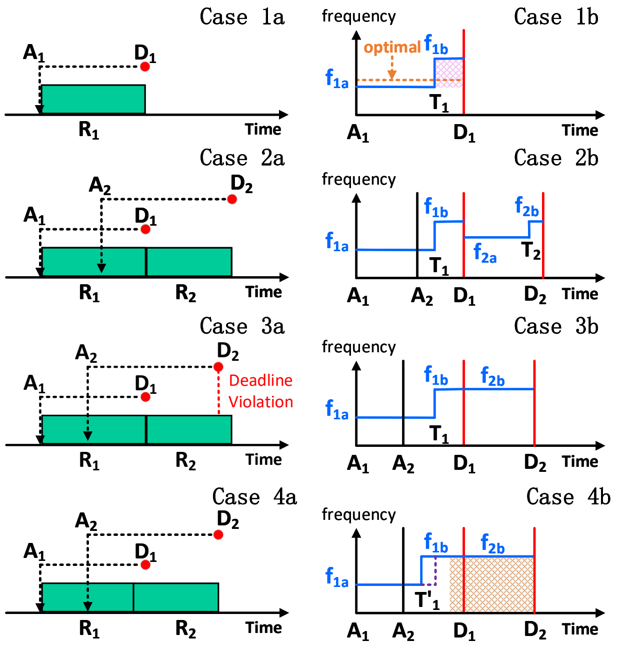


Figure 4: Swan design overview.

$$S_1^* = Predict^{LR}(Q_1, I | f_d) \tag{1}$$

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 posting list length. All the predicted service times are conditioned by the default CPU frequency f_d. For simplicity, we assume every request's amount of computation needed C (i.e., CPU cycles) and the work done for memory accesses M (i.e., memory cycles) don't change. A request's total work W is the sum of its CPU cycles C and memory cycles M. But the service time, S of the CPU is inversely proportional to the frequency f. Therefore, $W = C + M = f * S$ [2, 5]. The time delay for the CPU to transition from one frequency to another during with the CPU stalls, 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 should be constant (i.e., the dotted line in Fig. 4 *Case 1b*). This optimal frequency f_1 can be calculated as follows:

$$f_1 = S_1^* * f_d / (D_1 - A_1 - T_{dvfs}) \tag{2}$$

However, a prediction is likely to not be 100% accurate. Accounting for this, we have the following:

$$S_1^* = S_1 + E_1 \tag{3}$$

, where E_1 is the prediction error. With unknown S_1 during runtime, a step-wise DVFS control can produce better power savings [7]. For a two step DVFS as shown in Fig. 4 *Case 1b*, we have to solve three problems: 1.) select the initial frequency f_{1a} 2.) determine the time T_1 for frequency boosting and 3.) choose the boosted frequency f_{1b}. In Swan, we intuitively use the predicted service time S_1^* to select the f_{1a} as we want the initial frequency f_{1a} to be close to the optimal frequency f_1. Then, f_{1a} is:

$$f_{1a} = S_1^* * f_d / (D_1 - A_1) \tag{4}$$

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. 4 *Case 1b* would be straight (to D_1). Nevertheless, the shaded area reflects period we have to boost to f_{1b} to accommodate for prediction errors so that we meet the latency requirement. To find the correct value of T_1, we choose f_{1b} to be equal to f_d since we want the CPU frequency to stay at the lower f_{1a} for as long as possible. Since we do not know S_1 precisely during the runtime, a moving average of prediction errors is maintained and updated upon every request departure.

$$E_{avg} = \alpha * E_{old} + (1 - \alpha)|E_{current}| \tag{5}$$

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

$$f_{1a} * (T_1 - A_1) + f_{1b} * (D_1 - T_1 - T_{dvfs}) = (S_1^* + E_{avg}) * f_d \tag{6}$$

Notice that we execute at frequency f_{1b} for a time interval $D_1 - T_1 - T_{dvfs}$, since the CPU will stall for T_{dvfs} whenever we change the frequency. By combining equations 4 and 6, 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.

Two Requests. As shown in Fig. 4 *Case 2 and 3*, the scenario when at most two requests might stay in the queue can be more complicated as we need to re-configure T_1 to guarantee the request R_2's latency requirement. When the inter-arrival time $A_2 - A_1$ is large enough in *Case 2b* such that request R_2 can finish its job during 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. However, request R_2 in *Case 3a* will violate its deadline when the interval D_1 to D_2 is too short. In *Case 3b*, request R_2 is considered to be a critical request when the following happens:

$$(D_2 - D_1) * f_{2b} < (S_2^* + E_{avg}) * f_d \tag{7}$$

$W_2^e = (S_2^* + E_{avg}) * f_d$ is the total amount of predicted work 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_d). Then, 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^e. In our design, we always run a critical request at f_{2b} all the time.

In order to guarantee R_2's latency constraint in *Case 4b*, we have to re-configure T_1 to an earlier point, $T_1^{'}$, in order to finish R_1 early. Then, request R_2 (i.e., orange shaded area in *Case 4b*) can begin to execute even before D_1. Notice that we can only boost earlier 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 4b* is already f_d, when A_2 is between the initial value of T_1 and D_1. The adjusted boosting time $T_1^{'}$ can be obtained from:

$$f_{1a} * (T_1^{'} - A_1) + f_{1b} * (D_2 - T_1^{'} - T_{dvfs}) = W_1^e + W_2^e \tag{8}$$

, where we must finish the total work for the two requests – $W_1^e + W_2^e$ – before D_2. Notice that $T_1^{'}$ is always earlier than T_1. So, we don't need to worry about request R_1's deadline requirement as it will finish earlier because we change the frequency earlier than T_1. A special scenario of *Case 4b* is when an incoming request R_2 cannot finish its work even if we boost the CPU frequency to f_d immediately after it arrives (i.e., $T_1^{'} = A_2$). In such a case, it is safe to just directly drop request R_2, in the interest of saving more energy.

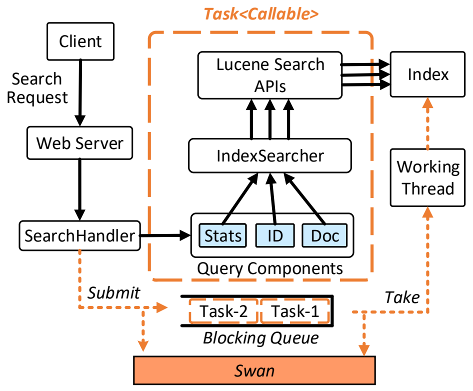


Figure 5: Swan implementation in the Solr search engine

N Requests. We now address the general case. Suppose that there are $N - 1$ requests in the queue and request R_k is the critical request among them. Similar to the case with two requests, nothing needs to be changed if the next request R_N is non-critical. If request R_N is critical, there are $N-1$ options to adjust the current frequency plan to complete the request R_N just-in-time. In Swan, our greedy algorithm tries to boost request R_{N-1} frequency earlier to see if request R_N can finish before D_N. If that adjustment cannot result in meeting the deadline, then we keep moving T_j earlier, towards D_{j-1}, in decreasing order of j until j equals 1. Request R_N will be dropped when all N requests in the queue are set to be executed at f_d, and still R_N is unable to meet its deadline. Assuming that T_m is selected to be changed, its value can be obtained as:

$$f_{ma} * (T_m^{'} - A_m) + f_{mb} * (D_N - T_m^{'} - T_{dvfs}) = \sum_{r=m}^{N} W_r^e \tag{9}$$

4 SWAN IMPLEMENTATION

Swan is implemented on the Solr search engine. Fig. 5 shows the essential parts of an ISN that are directly related with the Swan implementation. The Solr search engine contains three major parts: a web server, a Solr ISN instance and the underlying Lucene index searching APIs. The web server accepts a client's search request and forwards it to the SearchHandler. The SearchHandler searches for the query over the ISN index. All search queries are finally served by the IndexSearcher which calls the Apache Lucene APIs. The entire Solr search engine is written in Java. To implement Swan, we wrap the Query Components, IndexSearcher and Lucene Index Searching in Fig. 5 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 has only one working thread as we focus on single core power management. Finally, we conduct the ISN power management on submission or completion of each task.

As Swan is implemented as a part of the ISN, we leverage user-space DVFS control mechanisms. Swan leverages the Advanced Configuration and Power Interface (ACPI) to update the CPU core's

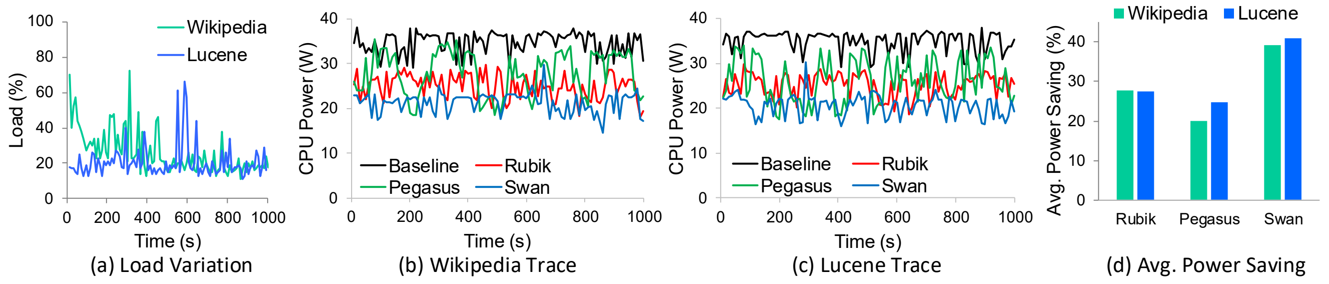


Figure 6: CPU power consumption results for Wikipedia and Lucene traces.

frequency at runtime. To reduce transition overheads, our frequency enforcement is achieved by manipulating each core's "scaling_setspeed" file. When this device file is changed, Linux triggers a group of system calls that take only 40 microseconds totally to update the CPU core's frequency. Thus, the extra latency overheads from the Swan implementation are negligible.

Our experimental setup has two machines connected by a 1G Ethernet link: one as the client and the other as the search engine server. The server machine is a 12-core Intel Xeon E5-2697 CPU, 128G memory running CentOS 7 operating system. On the client side, we wrote a Python program to replay our real search query traces. Two representative query traces are used in our experiments: a Wikipedia [13] trace and the Lucene nightly benchmark [4]. As only the Wikipedia query trace has request arrival timestamps, we use the same inter-request arrival pattern seen with Wikipedia for our Lucene nightly benchmark evaluations as well. On the server side, we deploy 12 single-working-thread based Solr ISNs on a 12-core CPU chip. Thus, each ISN is bound to one core of the CPU chip. The 12 ISNs receive the same search query from our Solr aggregator but schedule their core's frequency independently. In the search engine, we index the complete dump of entire English Wikipedia web pages taken on December 1st, 2018. This 65GB index has a total of 34 millions documents, which are uniformly distributed across the 12 ISNs. The search engine node supports per-core frequency scaling and per-socket voltage scaling. The CPU frequency can be selected over the range of 1.2 GHz to 2.7 GHz. Also, we disabled the CPU frequency Turbo Boost, and the default CPU frequency is 2.7 GHz (i.e., no power management). For power measurement, our CPU has sensors to monitor per socket's energy consumption with negligible overhead. 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.

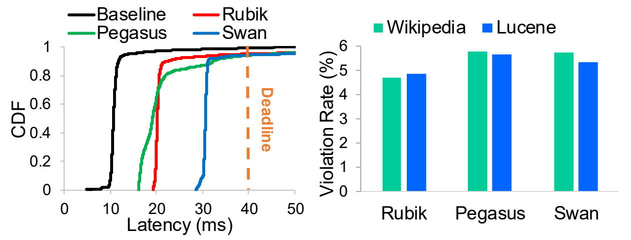
5 EVALUATION RESULTS

To evaluate the effectiveness of Swan, we implement Rubik [5] and Pegasus [6] on our infrastructure and compare our framework on two representative query traces. Rubik is an analytical, per request-based power management scheme, which utilizes a request's tail latency (e.g., 95th%tile) to select CPU frequency. On the other hand, Pegasus is a feedback based design, which measures the request's latency periodically and selects the highest CPU frequency if a deadline violation happens. When the measured latency is smaller than 65% of given time budget, the CPU frequency is reduced [6]. We will not compare our results to some other techniques that are based on CPU sleep states [9, 10] because ours is a DVFS strategy. Also, it is known that the deep state (C6) transition times are much higher than the DVFS.

Power Saving. Fig. 6 shows the CPU power consumption results under the Wikipedia and Lucene nightly benchmark query traces. The time series for the load variation for the two traces is in Fig. 6 (a). The ISN server load varies from 13% to 72% and the corresponding CPU power consumption is in Fig. 6 (b) and (c). Due to server load variations, the baseline with 2.7 GHz CPU frequency consumes 29.03 - 18.18 Watts for CPU power when running each of the two query traces. Pegasus results in the highest CPU power consumption, since its design is based on the feedback of deadline violations. It increases the CPU frequency immediately after a deadline violation, but decreases the frequency gradually, only after the measured latency is smaller than 65% of the given time budget. This hysteresis in reducing the frequency after a deadline violation means that some energy saving opportunities are lost. Next, we observe that Rubik consumes less CPU power than Pegasus. Rubik uses the tail of the service time distribution to guide the selection of the CPU frequency, striving to avoid even a low rate of deadline violations. Because it is so conservative, Rubik wastes a large amount of energy saving opportunities.

Instead, Swan boosts the frequency before the deadline anticipating a possible violation. Hence, the results in Fig. 6 (b) and (c) show that Swan has the least CPU power consumption at most of time. The reasons for this better energy efficiency are: a) the query specific latency prediction; and b) the two-step DVFS control. We must point out that Swan consumes more power than Pegasus at a limited number of data points due to the randomness of errors in latency prediction. To account for these prediction errors and meet request deadlines, Swan has to reserve a little latency slack for prediction errors. With improved prediction accuracy from a more sophisticated model, our design can consume even less CPU power. Fig. 6 (d) depicts the average CPU power savings for both Wikipedia and Lucene query traces across the three approaches compared to no power management. On the Wikipedia trace, the average power savings for Pegasus and Rubik are 20.07% and 27.78%, respectively. Swan performs the best with an average power saving of 39.08%. For the Lucene trace, we see similar results, with Swan achieving 40.85% CPU power saving.

Latency Distribution. In Fig. 7 (a), we plot the latency distribution for search requests with the Lucene query trace (results for the Wikipedia trace are similar). Request latencies are measured on one ISN. Similar to prior work [2, 5, 6, 10], the request deadline



(a) Latency Distribution (b) Violation Rate

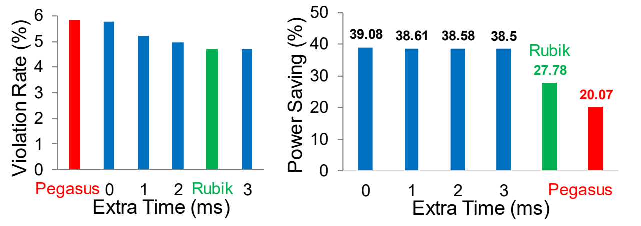
Figure 7: (a) Latency distribution for Lucene trace. Results for Wikipedia trace are similar. (b) Deadline violation rates.

is defined as the tail latency (e.g., 95th%tile in our experiments) of the baseline. As shown in Fig. 7 (a), the request latency under the baseline policy is around 10ms, while the request's time budget is 40ms. We observe that the "knee" of the latency distribution for the baseline moves from around 10ms to 16.2ms with the epoch-based Pegasus. With its per request frequency selection, Rubik shifts the latency distribution "knee" closer to the deadline. As we have a query specific latency predictor, most of the requests under Swan are closer to their deadlines, in comparison with Rubik and Pegasus. For all the power management schemes in Fig. 7 (a), there is still a large gap between the "knee" of latency distribution and deadline. The reason is that all the schemes have to guarantee that at least 95% of the search requests meet their deadline. One possible solution to reduce this gap is to develop a multi-feature model [8] in Swan with high prediction accuracy. This is work we plan for the near future.

Fig. 7 (b) reports the deadline violation rates of Rubik, Pegasus and Swan with two different query traces. All three frameworks achieve around 5% deadline violations on both traces, with Rubik having the least – 4.7% – on the Wikipedia trace. The deadline violations in Swan are due to inaccurate service time prediction. In Section 3, we use a moving average E_{avg} to quantify the prediction error. Besides improving the predictor's accuracy, a simple way to reduce Swan's deadline violation rate (shown in Fig. 7 (b)) is to add an extra time to E_{avg} (i.e., results in increasing the estimate of residual work, hence service time estimate, so the frequency is boosted earlier). The impact of adding the extra time to E_{avg} on deadline violation rate and average power saving for Wikipedia trace are given in Fig. 8. In this experiment, we keep increasing the estimate of the prediction error by 1ms each step. In Fig. 8 (a), the violation rate without any extra time to E_{avg} is 5.76%, which is a bit better than Pegasus (5.8%). After adding 3ms, Swan is better than Rubik (4.7%), by going down to a 4.68% violation rate. This is a reasonable trade-off, as we see in Fig. 8 (b) that the corresponding average power saving of Swan deteriorates only by 0.58%. Mainly, the average power saving of Swan can still be quite high, at 38.5%, much better than Rubik's 27.78%.

6 CONCLUSION

We present Swan, a two-Step poWer mAnagement for distributed search eNgines, that achieves significant power savings even when the accurate query-specific service time is unknown. Swan utilizes a simple linear model to roughly estimate each request's service time and then selects an initial CPU frequency for power optimization. To minimize deadline violations due to prediction errors, we boost the



(a) Deadline Violation (b) Avg. Power Saving

Figure 8: Sensitivity results for E_{avg} estimation in Swan (blue bars).

initial frequency at the right time to catch-up on a request's deadline. Further, the boosting time of a current request is re-configured on the arrival of each critical request so that we meet its latency constraints. Swan is implemented in the Solr search engine and evaluated on a real multi-core server, achieving at least 39% CPU power saving on average.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient Query Evaluation Using a Two-level Retrieval Process. In *Proceedings of the CIKM '03*. ACM, 426–434.
[2] C. Chou, L. N. Bhuyan, and D. Wong. 2019. μDPM: Dynamic Power Management for the Microsecond Era. In *2019 IEEE HPCA*. 120–132.
[3] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the ASPLOS '15*. 161–175.
[4] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the MICRO-50 '17*. ACM, 625–638.
[5] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-Critical Systems. In *Proceedings of the MICRO-48 '15*. ACM, 598–610.
[6] 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 ISCA '14*. IEEE Press, 301–312.
[7] Jacob R. Lorch and Alan Jay Smith. 2001. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proceedings of the SIGMETRICS '01*. ACM, 50–61.
[8] Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. 2012. Learning to Predict Response Times for Online Query Scheduling. In *Proceedings of the SIGIR '12*. ACM, 621–630.
[9] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: Eliminating Server Idle Power. In *Proceedings of the ASPLOS '09*. ACM, 205–216.
[10] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: Architectural Support for Deep Sleep. In *Proceedings of the ASPLOS '12*. ACM, 313–324.
[11] Luo Si and Jamie Callan. 2003. Relevant Document Distribution Estimation Method for Resource Selection. In *Proceedings of the SIGIR '03*. ACM, 298–305.
[12] Paul Thomas and Milad Shokouhi. 2009. SUSHI: Scoring Scaled Samples for Server Selection. In *Proceedings of the SIGIR '09*. ACM, 419–426.
[13] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Comput. Netw.* 53, 11 (July 2009), 1830–1845.
[14] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the MICRO-48 '15*. ACM, 585–597.
[15] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. 2004. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *Proceedings of the ASPLOS '04*. ACM, 248–259.
[16] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan. 2019. Goldilocks: Adaptive Resource Provisioning in Containerized Data Centers. In *2019 IEEE ICDCS*. 666–677.
[17] L. Zhou, C. Chou, L. N. Bhuyan, K. K. Ramakrishnan, and D. Wong. 2018. Joint Server and Network Energy Saving in Data Centers for Latency-Sensitive Applications. In *2018 IEEE IPDPS*. 700–709.